MODELS FOR DATA SOURCE TRACING

WITH XML

THESIS

Teoman YORUK, 1st Lt., TUAF

AFIT/GCE/ENG/01M-05

DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

APPROVED FOR PUBLIC RELEASE, DISTRIBUTION UNLIMITED.

MODELS FOR DATA SOURCE TRACING WITH XML

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science

Teoman YORUK, B. S. Computer Engineering

1st Lt., TUAF

March 2001
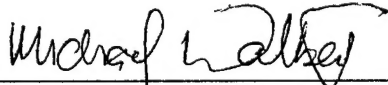
MODELS FOR DATA SOURCE TRACING WITH XML

THESIS

Teoman YORUK, B.S. Computer Engineering
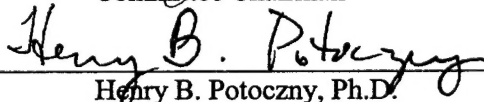
1st Lt., TUAF

Approved:

Michael L. Talbert	5 Mar 2001
Michael L. Talbert, Ph.D., Major, USAF	date
Committee Chairman	
Henry B. Potoczny	5 March 2001
Henry B. Potoczny, Ph.D.	date
Committee Member	
Karl S. Mathias	5 MARCH 2001
Karl S. Mathias, Ph.D., Major, USAF	date
Committee Member	

ACKNOWLEDGMENTS

TABLE OF CONTENTS

TABLE OF FIGURES

ABSTRACT

The Air Force Research Laboratory, Sensors Directorate, Electronic Warfare Simulation Branch (AFRL/SNZW) is responsible for developing and maintaining real-world and hypothetical scenarios for an array of threat engagement simulation systems. The general process for scenario creation involves mapping from real-world databases and operations plans to specific fields in the input files which represent the scenario.

As part of the AFRL/SNZW's overall initiative for the development of a Collaborative Engineering Real-Time database CORrelation tool (CERTCORT), as important to the scenario files themselves is the capability to trace back to the source of data for the scenario fields. Acquiring this capability consequently results in the reusability of the old scenario components.

In this research, a nascent markup technology, eXtensible Markup Language (XML) and its derivative languages are studied as a basis for representing and capturing the source of data for the fields of an old scenario file and exploiting it for the creation and editing of new scenario files.

MODELS FOR DATA SOURCE TRACING WITH XML

1 INTRODUCTION

1.1 BACKGROUND

Although the cold war is over, there is still a need for the armed forces. There is no specific threat as before, but because of the volatility of human nature, people do not trust other people. Most of the countries' defense concept is to lessen the number of military personnel, have more precise and effective weapons, and make their units mobile- ready to take action any time, anywhere. Military budgets have always been a point of dispute for all countries. As long as there is no war or real threat, people think that the money is spent in vain for men who are waiting idle and producing nothing but consuming all kinds of resources. So armed forces have to lessen their expenses.

To be successful in military operations and not to experience unexpected results in the theatre of war, generals need to know the capabilities of available manpower and weapon systems in advance. Real tests and real exercises are too expensive, so running simulations is inevitable. In fact, besides being cheaper, simulations are much safer, and if people make mistakes they can make up for it.

In order to run simulations, a great amount of source data is needed. The data source can be in the format of databases or text files or any marked-up document. Out of that great data source, a seemingly infinite number of real-world scenarios can be modeled.

This information mapping from databases or text documents to scenario files and being capable of tracing back source of information for the elements of the scenario file are, in fact, vital parts of the simulations. Because the more consistent and appropriate data you have in the scenario files, the more accurate results you will get from running simulations.

The Electronic Combat Modeling Branch of the Sensors Directorate of the Air Force Research Laboratory (AFRL/SNZW) at Wright Patterson AFB, OH develops and supports real

world and hypothetical scenarios for an array of threat engagement simulation systems. The general process for scenario creation involves mapping from real-world databases and operations plans to specific fields in the input files which represent the scenario in Figure 1 [18].
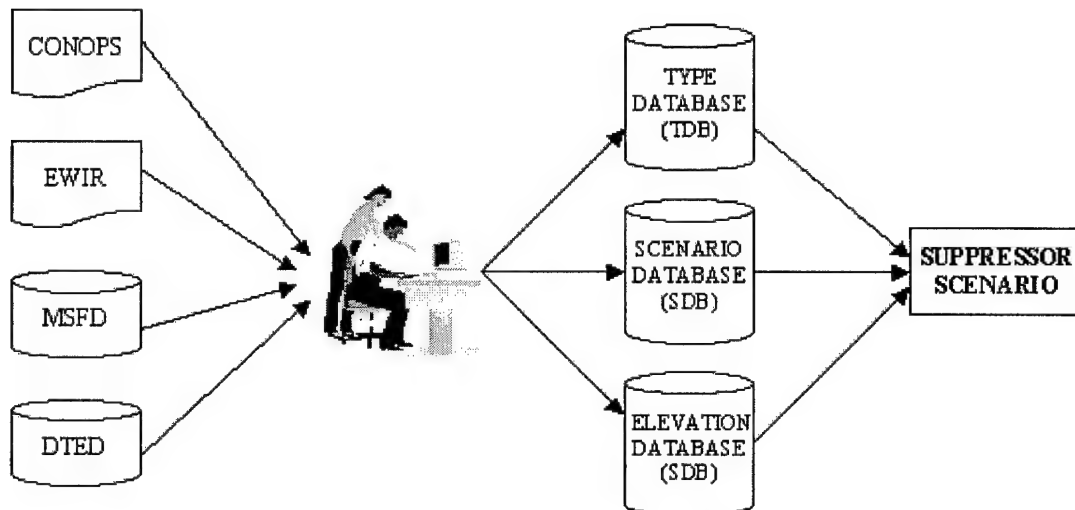
Figure 1- Source Data to Scenario File Mapping.

1. 2 PROBLEM DESCRIPTION

Previous work at the Air Force Institute of Technology, School of Engineering and Management [18] has focused on parsing and processing the text-based scenario files associated with the Suppressor combat simulator. McDonald's work did not include maintaining traceability from originating files or documents. As part of the AFRL/SNZW's overall initiative for the development of a Collaborative Engineering Real-Time database CORrelation Tool (CERTCORT -- Figure 2 [18]) as important to the scenario files themselves is the source-to-scenario mapping, especially when the sources are text documents, as depicted in Figure 2 (ConOps and other documents).
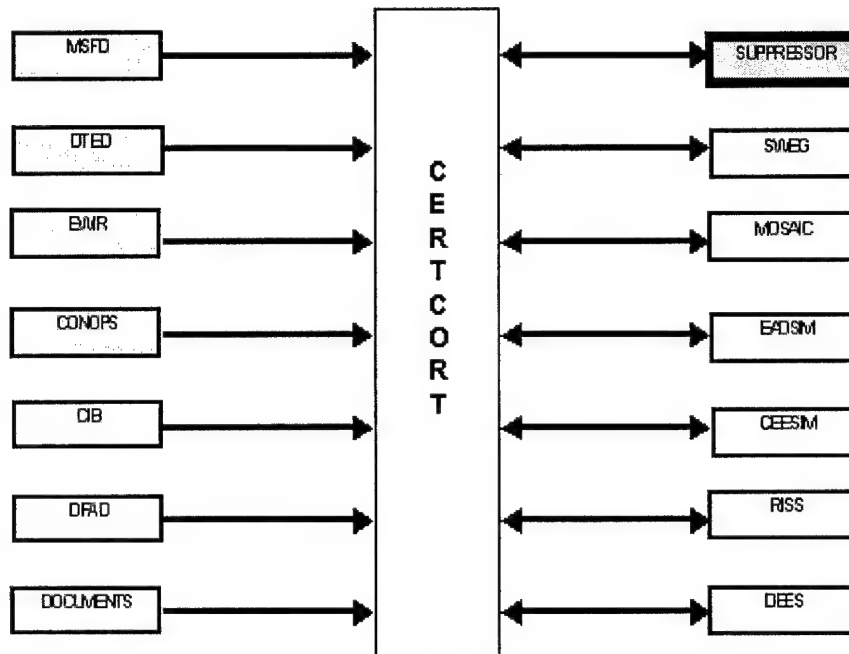
Figure 2- CERTCORT Architecture.

Currently, the process of mapping data input files is done by hand, is very tedious, and is done by one analyst at a time [18]. An overriding goal of the CERTCORT system is to allow scenario input source files that describe weapon system parameters and behavior (MSFD, DTED, CONOP, EWIR, and so on) to be introduced dynamically without changing the underlying information architecture of the system. This idea is clearly stated in [42] as follows:

"The ability to trace data input files to their corresponding application within a given scenario is key to building the collaborative simulation scenario environment that is the CERTCORT vision."

The analyst traverses back to the data source and feeds data into scenario fields depending upon the information he gets from the data source. He manipulates and changes the fields of the scenario file but never the source of data.

The CERTCORT system should also allow the information contained in any source input file to be incorporated in scenarios of any model belonging to the CERTCORT environment (SUPRESSOR, JIMM/SWEG, EADSIM, etc.).

3

1. 3 RESEARCH FOCUS

The focus of this research is to examine how the combination of Java and eXtensible Markup Language (XML) technologies can be applied to the CERTCORT problem domain. Particularly, to explore the use of eXtensible Markup Language (XML) as a basis for representing and capturing the source of data and exploiting it for the creation and editing of scenario files.

In order to make a text document describe itself in a way software applications can "understand," the data in the document has to be marked-up. So marking up the data source with XML allows one to access portions of the document related to scenario parameters. However, data in an XML-based document is only static, and to be accessed for use in CERTCORT, it needs to be processed by tag-matching applications. Java technology is a good match for XML, because it offers the portable code which the portable data provided by XML technology needs to be used in a completely platform independent manner. (Further information on XML and Java is given in Chapter 2.)

One type of scenario file (TDB) and one type of data source input file (which is supposed to represent Multi-Spectral Force Deployment Database (MSFD)) are analyzed by this research. The main goal is to show how dynamic flow of information from a data source input file, e.g., MSFD, to a scenario file, e.g., Type Database (TDB), is attained and how these data can be changed by going back to the origin of the data so that the scenario file can be reusable and flexible.

Chapter 2 provides sufficient background to provide a basic understanding of XML technology, the benefits of using XML, how it can be combined with Java and finally why it has been chosen as a mark-up language for this research.

Chapter 3 presents the methodologies developed to trace the source of data for a scenario file represented by an object model, and how to manipulate and create new scenario files using this model. Chapter 4 describes the results of implementing the methodologies described in

4

Chapter 3. Chapter 5 presents a brief demonstration of the software components implemented in Chapter 4. Finally, conclusions and recommendations for future work are discussed in Chapter 6. Appendices are included to provide detailed information on the link builder model introduced in section 3.4.2.

1.4 LIMITATIONS

Before proceeding to Chapter 2, the last point to be mentioned is this: As a result of being an officer of Turkish Air Force (TUAF), I do not have the authorization to view the classified United States Air Force (USAF) documents and much of the ground-truth data associated with this research area is contained in documents, not accessible to this author.

The solution proposed by senior USAF officers and my thesis advisor is to prepare some dummy documents which represent the scenario file (TDB) and the source data input file (MSFD). As a result the contents of the scenario files and the data source ground truth files are not the actual ones. I do not consider it as a disadvantage, because the main goal of the research is to develop a mechanism which allows one to trace back to source data, or to put it in other words *dynamic information traceability*. Due to the fact that the source and scenario files are dummy, the mapping between them is also dummy, though the mechanism is the same either way.

1.5 SUMMARY

In this chapter the background for this research, the problem which forms the topic of this research, preceding works, assumptions and limitations related to this research have been introduced. In the next chapter, the technologies which play an important role in solving the research problem are introduced.

2 LITERATURE REVIEW

2.1 INTRODUCTION

Computers are not capable of extracting semantic data from "flat" or "non-marked up" documents, unless the documents themselves do provide some information about their contents. The information to tell something to computers about the semantics of a document's contents can be provided by mark up languages. And the fundamental markup technology chosen for this research in order to help trace information is the Extensible Markup Language (XML).

But is this enough? No. There is also need for an application programming language which is able to parse such a document and allow one to manipulate the contents of a document. Simply put this is because an XML document, by itself, is not "active", but only presents a hierarchical view of the semantic units of a document. The application technology chosen manipulating XML-based documents for this purpose is Java, by Sun Microsystems, Inc.

This chapter introduces a basic understanding of XML, XML history, how XML and Java technologies can be combined to trace information and other uses of XML for intermediate data representation.

2.2 XML HISTORY

The Extensible Markup Language (XML) emerged in 1998 as a next generation technology for structuring and exchanging information on the World Wide Web (WWW). It was officially accepted by the WWW Consortium (W3C), an international consortium for proposing standard technologies for the WWW, in the same year. It has been center of great attention since then, because it promises many new possibilities that were not available in other markup languages.

The more popularly known Hypertext Markup Language (HTML), which is a sibling to XML, has similarities with XML. It also tags the contents or components of the data in a document, for example;

<TITLE> The Weather Forecast </TITLE>

<H1> Humidity: 67% </H1>

But, the tags used in an HTML document tell the parsers how to present or visualize the data between the tags, and the tags do not express any semantics about the data content. In XML, it is possible to create user-defined tags which can denote semantics related to the content inside the tags. In other words, tags can tell what kind of data is stored between the start tag and end tag. That is making the data self-describing. If the HTML document fragment given above is rewritten using XML, it looks as below;

<Weather> The Weather Forecast </Weather>

<Humidity> 67% </Humidity>

The important point to mention about XML document is that, in the second line since the tag "<Humidity>" reflects the idea that humidity data is stored between the tags, there is no need to write "Humidity" again inside the tags, as is the case with HTML document fragment. The tags are created by the document builder, without being dependent on any set of rules defined by the markup language. The advantages of XML over other markup languages is discussed in more detail in the section "Benefits of XML".

In fact it is not right to characterize XML as a completely new technology, because XML is a subset of the Standard Generalized Markup Language (SGML). It has the key SGML advantages extensibility, structure and validation. XML is designed to be easier to learn, use, and implement than SGML. XML adheres to the Standard Generalized Markup Language (SGML) specification, which has been an ISO standard since 1986. [10]

With the advantages it provides, XML is likely to be the main markup language to tag, process, transform and exchange documents for the foreseeable future.

2.3 XML DOCUMENT STRUCTURE

An XML document typically presents a document's contents hierarchically. This hierarchy can be regarded as a tree structure, which has a single root at the top. The root contains children elements, and in a nested fashion the children can contain other children elements. The XML document elements have relations like *sibling, parent, child,* (in the same way as any tree structure.) Figure 3 depicts a simple XML document and Figure 4 depicts how this XML document can be converted to a tree structure.

```
<?xml version="1.0"?>
<documentlist>

  <required_info>
    <angle id="some"> 45 degrees below 10000 feet </angle>
    <speed id="some"> The min speed required is 700 KNOTS </speed>
  </required_info>

  <optional_info>
    <angle_op id="never">13 degrees below 10000 feet </angle_op>
    <speed_op id="never"> The min speed is 675 KNOTS </speed_op>
  </optional_info>

</documentlist>
```

Figure 3- A Simple XML document.
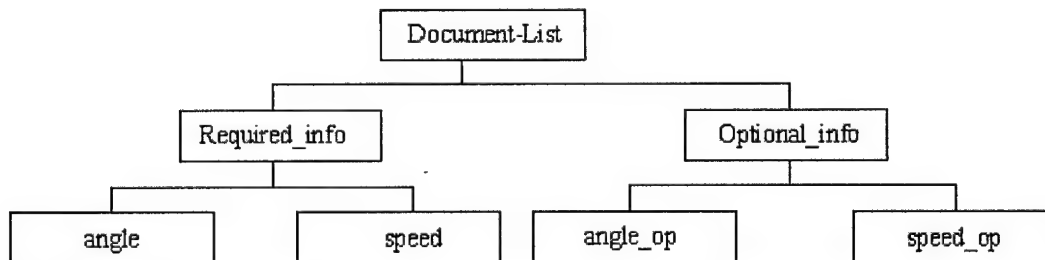
Figure 4- Tree representation of the simple XML document.

2.3.1 Testing the Structural Correctness of an XML Document

When building XML documents with a text editor, it is possible to make structural or syntactic mistakes. There are two independent steps for checking the grammar of an XML document. First step is checking to see if a document is *well-formed* and the second and a more complete step is checking to see if a document is *valid*.

It is XML parser's job to determine an XML document conforms to the grammar specified by XML Specification 1.0. This grammar is called as "Extended Backus–Naur Form (EBNF)" which refers to a "set of acceptable token sequences, which in turn defines the syntactic correctness of a statement in a language." [3]

2.3.1.1 Well-Formed Document

To see if a document is well-formed, it is checked against EBNF. That is, the parser checks such rules as, every start tag has an ending tag, all the words are written syntactically correct, etc.. For example, a mistake shown in Figure 5 makes a document not well-formed.

```
<?xml version="1.0"?>
<documentlist>

    <required_info>
      <angle id="some"> 45 degrees below 10000 feet </angle>
      <speed id="some"> The min speed required is 700 KNOTS </speed>
    </required>◄················    Incorrect End Tag

    <optional_info>
      <angle_op id="never">13 degrees below 10000 feet </angle_op>
      <speed_op id="never"> The min speed is 675 KNOTS </speed_op>
    </optional_info>

</documentlist>
```

Figure 5- Not a well-formed XML document.

When the document shown in Figure 5 is parsed, the parser gives the error "end tag does not match the start tag". Because, there is an inconsistency for "required_info" element's start and end tags. The correct expected end tag is "</required_info>".

2.3.1.2 Valid Document

Checking the validity of a document is more complex than checking the well-formedness of a document, because in addition to EBNF, the document is checked against a Document Type Definition (DTD). DTD is to an XML document what a schema is to database. It is metadata which describes the structure and thus the hierarchical relations of elements of a document. Figure 6 shows the valid XML document and its DTD which was depicted in Figure 3 (without a DTD) as a well-formed document.

```
<?xml version="1.0"?>
<!DOCTYPE documentlist [
  <!ELEMENT documentlist (required_info, optional_info)>
  <!ELEMENT required_info (angle, speed)>
  <!ELEMENT optional_info (angle, speed)>
  <!ELEMENT angle (#PCDATA)>
  <!ELEMENT speed (#PCDATA)>
  <!ATTLIST angle id CDATA #REQUIRED>
  <!ATTLIST speed id CDATA #REQUIRED>
]>
<documentlist>

  <required_info>
     <angle id="some"> 45 degrees below 10000 feet </angle>
     <speed id="some"> The min speed required is 700 KNOTS </speed>
  </required_info>

  <optional_info>
     <angle_op id="never">13 degrees below 10000 feet </angle_op>
     <speed_op id="never"> The min speed is 675 KNOTS </speed_op>
  </optional_info>

</documentlist>
```

Figure 6- A valid XML document and its DTD.

The tag layout of the XML document is checked against the grammar described by the associated DTD. If the structure in the XML document does not match the nesting, rules and

10

syntax defined in the DTD, the parser used is going to give errors. Figure 7 depicts an invalid XML document.

```
<?xml version="1.0"?>
<!DOCTYPE documentlist [
  <!ELEMENT documentlist (required_info, optional_info)>
  <!ELEMENT required_info (angle, speed)>
  <!ELEMENT optional_info (angle, speed)>
  <!ELEMENT angle (#PCDATA)>
  <!ELEMENT speed (#PCDATA)>
  <!ATTLIST angle id CDATA #REQUIRED>
  <!ATTLIST speed id CDATA #REQUIRED>
]>
<documentlist>

  <weather>
     <angle id="some"> 45 degrees below 10000 feet </angle>
     <speed id="some"> The min speed required is 700 KNOTS </speed>
  </weather>

  <optional_info>
     <angle_op id="never">13 degrees below 10000 feet </angle_op>
     <speed_op id="never"> The min speed is 675 KNOTS </speed_op>
  </optional_info>

</documentlist>
```

Figure 7- An invalid XML document

The XML document above, in fact, is well-formed, it obeys the rules defined by EBNF, but it does not obey the rules defined by the DTD. Since there is no such element as "weather" defined in DTD, "weather" element can not be added to the structure of this XML document. It is possible to keep the DTD in a separate file (external DTD) or it can be included at the very beginning of the XML document before the actual tagging starts (internal DTD) as shown in Figure 7. If an external DTD is used, there has to be a reference to where the external DTD is stored so that the application parsing the XML document can find DTD. The reference is typically a URL or file name as in Figure 8.

```
<?xml version="1.0"?>
<!DOCTYPE DOCUMENT SYSTEM "c:\mydocuments\ch14\sample.dtd">
<DocumentList>
            This figure depicts how an external DTD is defined in an XML document.
</DocumentList>
```

Figure 8- How external DTD is defined

In this example, sample.dtd is in an external file and the address to the external DTD may be defined as shown above. Most parsers available require a DTD, like Sun's XML parser for Java (JAXP 1.0) or IBM XML parser for Java. But there are also parsers which do not require a DTD, such as Aelfred XML parser available from Microstar Software, Ltd. [49].

2.4 BENEFITS OF XML

The basic features of XML which make it superior to other markup languages are as follows:

Simplicity and Structure: As discussed in the section "XML Document Structure," XML represents data in a hierarchical way. It can also be expressed as a tree. The uncomplicated structure of the language allows one to model data to any level of complexity. The structure maintained by the language becomes especially important when one needs to extract data from the document or manipulate the document.

Extensibility: Contrary to other languages like HTML, XML allows the document builder to define his own tags. This gives independence and flexibility when building documents. The document builder can easily define new tags of his own and add them to the document structure when needed.

Validity: Using a DTD, the document builder can define his set of rules used to build the document. This issue is especially important when it comes to exchanging documents among organizations or on the Web. The receiving parties can understand the document's structure by checking its DTD, and then correctly extract the data for inclusion into their databases or systems.

In a way, DTDs define the underlying *protocol* for exchange of data. DTD is something everybody agrees to obey.

Tagging the Meaning Rather than the Look of Data: XML tags have the ability to describe what is inside the tags. This adds semantics to the structure of the document and makes it self describing. This ability of XML is one of its features which makes it superior to other languages like HTML.

As expressed in [12] *"A browser can not query, sort or process in any way a set of rows from a database transformed into HTML, so the usefulness of information can not be extended beyond this display. The information about the columns that the data came from is lost. "*

Platform and Media Independence: The data represented in an XML document is language neutral; any programming language can read this data. Also the contents of the document can be published in various formats using XML Style Language (XSL) and script languages.

The main impetus for the design of XML was to exchange documents between organizations, and it is becoming clear that universal document exchange and data sharing will rely heavily on XML. [21]

Databases of Information: There are two answers for the question " Is XML a database?" When an XML document is considered alone, it is not a database because it is only a marked up text document; there is no way to manipulate or extract the data in that text document. But provided that the document is supported by applications which have the tools to parse the document and access its contents, then it can be considered as a database.

Wide Adoption: As discussed in [14] *"The next generation of the major Web browsers (most notably, version 5 of both Microsoft Internet Explorer and Netscape Communications Corp.'s Navigator) have robust, built-in XML support. Major and minor software vendors are announcing XML support daily, most notably Microsoft, with its e-commerce strategy based on XML. Many vertical industries have begun initiatives to create standards for data exchange based on XML. "*

13

Ease of Updating Documents: Think of a server and its clients having the same copies of a big XML file. When the server makes some changes on the XML file, it need not send the whole XML document to its client. Sending the portions of the file changed is enough to update the files on the clients. [4]

This can be achieved using Xpath. Xpath lets one to locate the exact portion of a document. It is similar to defining paths for directories like *"c:\mydocuments\thesis."* The path to reach a precise point in an XML document starts with its root and goes all way down to the desired portion. For example: Imagine a server responsible for keeping information of customers of a store selling shoes up to date. If the information about a customer is changed then the server need not send all the document to all of its clients. First the server defines the path for the part of the document to be changed like "customerlist/customers/private_information/home_address..." and together with the path, the changed portion is sent to the clients. The clients taking this message follow the path given and replace that portion of the document with the new one.

Precise and Fast Searches: As mentioned above, XML tags have semantics; they have the capability to describe the content of an element. This gives XML a great power in information retrieval (searching, indexing, locating and extracting information). Searches can be restricted to specific tagged parts of a document, the parts of the document matching the specified tag will be relevant. It also makes searching faster. For example, to list the temperatures of all the cities from an XML document containing weather reports of all the cities in a state, rather than searching the entire document, searching the <temperature> tags and extracting their contents is enough. Expounding on this capability, a remarkable point about the general concept of this research is wisely expressed in [21]:

*"Even that is just an intermediate step. Librarians figured out a long time ago that the way to find information in a hurry is to look not at the information itself but rather at much smaller, more focused sets of data that guide you to the useful sources: hence the library card catalogue. Such information about information is called **metadata**. "*

Advantages of Using XML on the Web: When all the capabilities of XML is widely used and everybody promises to play with the rules defined by XML specification 1.0, the Web will be faster, friendlier and a better place to do business.

XML makes Web publishing and content management easier. This is because of the flexibility provided by XML tags. Once documents are marked up with XML tags, the data from all these documents can selectively be combined into a single document. XML has a promising standard called XML Style Language Transformations (XSLT). XSLT allows one to manipulate the components of an XML document selectively. Also, the presentation, to the user, of the data inside these documents can be done by XSLT. Detailed information is given on XSLT in the following sections. Other benefits of using XML on the Web is clearly expressed in this example taken from [21]:

"Imagine going to an on-line travel agency and asking for all the flights from London to New York on July 4. You would probably receive a list several times longer than your screen could display. You could shorten the list by fine-tuning the departure time, price or airline, but to do that, you would have to send a request across the Internet to the travel agency and wait for its answer. If, however, the long list of flights had been sent XML, then the travel agency could have sent a small Java program along with the flight records that you could use to sort and winnow them in microseconds, without ever involving the server. Multiply this by a few million Web users, and the global efficiency games become dramatic."

2.5 KEY XML TECHNOLOGIES

XML presents data as a document, but can this data be managed in this plain form? It is great to have a document marked up complying with the grammar defined by a language, but the data inside the document is not activated yet and in this plain form it does not have the capability to tell much or do something on its own and produce an outcome.

15

Use of an XML document requires applications which are capable of accessing the contents of the document, extracting data from the document, manipulating the structure and the style (presentation) of the document. The XML APIs capable of doing the tasks listed above are of two types; Document Object Model (DOM) and Simple API for XML (SAX). Brief information on DOM and SAX is given next.

2.5.1 Document Object Model (DOM)

The term "object model" reminds people of a set of objects related to each other, presented in hierarchical fashion. In the case of an XML document, an object model means a graphical representation reflecting the structure of the entire XML document.

The **Document Object Model (DOM)** is an application program interface (API) for HTML and XML documents that allows programs to access and update the content of documents and manipulate their structure. What the DOM does is to facilitate the creation of an in-memory tree of objects that applications can traverse to extract information [3].

When the XML processor parses an input-stream, it maps the result to the DOM for further manipulation by the application. A tree representation of the document is built in memory. After the system has parsed the XML input into the DOM object nodes (hierarchical, parent/child node relationships), an application can access the objects for further processing.

One important target for the DOM is to provide a standard programming interface that can be used in a wide variety of environments and applications. The DOM is designed to be used with any programming language [30]. Consider the following XML document fragment:

```
<Customer_List>
    <Customer_Info>
        <F_Name> Bob </F_Name>
        <L_Name> Brown </L_Name>
        <Credit type = "permanent"> 5,000 USD </Credit>
    </Customer_Info>
</Customer_List>
```

Figure 9- An XML document fragment.

There is a hierarchical relationship inherent in the above code fragment. The figure below illustrates the in-memory DOM tree constructed from the above document fragment. The DOM tree has the same hierarchy as the document.
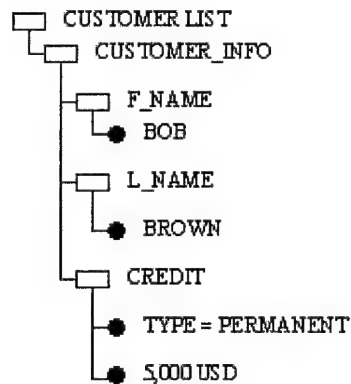
Figure 10- Tree Object constructed from the XML document fragment.

By using a DOM API, it is possible to create an empty XML document from scratch. What is done is to build a tree structure first, then populate the tree structure with data and finally write out the data in the tree structure as an XML file. This gives the ability to build XML documents dynamically.

The following 10 important DOM methods give a more detailed characterization of capabilities afforded by using a DOM API [27]:

- getDocType
- getName
- getElementsByTagName
- item
- getFirstChild
- getNodeValue
- getAttribute
- getChildNodes
- getLength
- getTagName

The combination of these methods to extract data from an XML document is demonstrated in Figure 11.

```java
public void load(Document d, String bb, String atbval) {
        Node node;
        NodeList childList, child1List;
        Element el = null;
        String vv = null;
        String aa = null;

        childList = d.getElementsByTagName(bb);

        if (childList.getLength() < 1) {
                System.err.println("There is no such tag in XML DOC");
                return;
        }

        for (int j = 0; j < childList.getLength(); j++) {
                el = (Element) childList.item(j);
                aa = el.getAttribute("id");
                System.out.println(aa);

                if (aa.compareTo(atbval) == 0) {
                        child1List = childList.item(j).getChildNodes(); //returns nodelist
                        vv = child1List.item(0).getNodeValue();
                        JFrame frame = new MakeTree11(el, vv, bb);
                        frame.show();
                }
        }
}
```

Figure 11- Java code fragment showing the usage of methods in DOM API.

In the Java code fragment given in Figure 11, "getElementsByTagName(bb)" method looks for the tag name assigned to the String variable "bb" in the XML document. All the matching elements are put in a list ("childList"). An iteration is made through this list by using "getAttribute("id")" and from the list of the selected elements, the element having the attribute value "atbval" is found.

XML specification 1.0 provides an other API which may be used instead of the DOM API. A brief elaboration on the Simple API for XML (SAX) which is an alternative mechanism to manipulate XML documents follows.

2.5.2 Simple API for XML (SAX)

Simple API for XML is an event driven approach. When parsing an XML document it regards the starting tags and ending tags as events. It makes use of a stack based algorithm. For transforming linear events into a tree structure, the tags encountered and the content between is put in a stack and the rule of last in first out (LIFO) is implemented when pulling data out of the stack. As a consequence, random access is not possible with SAX, though it is with the memory mapped DOM model, it allows sequential access. SAX is however, especially useful for large documents [2]. This is because when compared to DOM, SAX requires less memory.

2.5.3 XML Style Language (XSL)

One of the features of XML is that it separates the content and the style of a document. That is the content of a document is kept in file and its style which determines how the document is going to be presented is kept in another file.

An XSL style sheet is an XML document. A reference is made to the style sheet within content of the document. When there is a need to change how the document should look, simply changing the style sheet is enough. The output obtained from applying style sheet to an XML document can be directed into a file or piped into another application [4]. XSL specifies the styling of an XML document by using XSL Transformations (XSLT) to describe how the document is transformed into another XML document [41]. Figure 12 depicts how this is achieved.

Hierarchically represented XML data   XSL Transformation   New XML Data
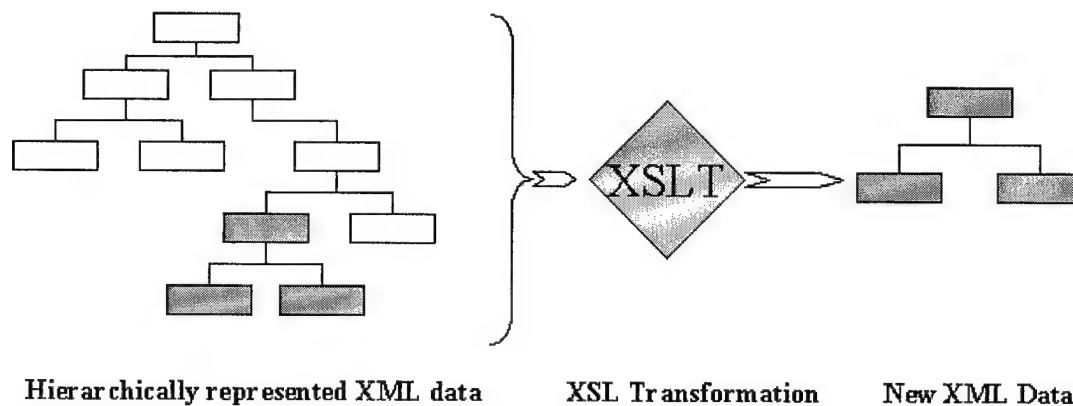
Figure 12 - How XSLT works with XML

As in the figure above, XSLT converts the source tree into a result tree. The resultant subtree is the part of the original XML document that is to be presented. XSLT makes use of the expression language defined by **Xpath** for selecting elements for processing, for conditional processing and for generating text [41]. Xpath is a language for addressing parts of an XML document, designed to be used by both XSLT and Xpointer [39], it uses a path notation as in URLs for navigating through the hierarchical structure of an XML document. The style sheet also contains information like color to be used and images to be presented to the user.

Figure 13 depicts a well-formed XML document which contains information about students. At the second line of the XML document in Figure 13, the style sheet which transforms the XML document and makes it presentable is defined, "tezola.xsl". The file path or the URL address to the style sheet can be defined in "href" part. "tezola.xsl" is presented in Figure 14.

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="tezola.xsl"?>
<studentlist>
    <student>
        <fname> Joe </fname>
        <lname> Brown </lname>
        <gpa> 3.4 </gpa>
    </student>
    <student>
        <fname> Carol </fname>
        <lname> Red </lname>
        <gpa> 3.7 </gpa>
    </student>
    <student>
        <fname> Beth </fname>
        <lname> Green </lname>
        <gpa> 3.0 </gpa>
    </student>
</studentlist>
```

Figure 13- A well-formed XML document

```
<?xml version='1.0'?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
<xsl:template match="/">
  <html>
  <body>
   <table border="2" bgcolor="yellow">
    <tr>
        <th>F_Name</th>
        <th>L_Name</th>
        <th>GPA</th>
    </tr>
    <xsl:for-each select="studentlist/student" order-by="+ gpa">
    <tr>
     <td><xsl:value-of select="fname"/></td>
     <td><xsl:value-of select="lname"/></td>
     <td><xsl:value-of select="gpa"/></td>
    </tr>
    </xsl:for-each>
   </table>
  </body>
  </html>
</xsl:template>
</xsl:stylesheet>
```

Figure14- Content of the "tezola.xsl"

The XML document in Figure 13 is styled by the rules defined by tezola.xsl above. First

thing noticed is that to present the data in the document, a table having the columns "F-name",

21

"L_name", "GPA" is created. Then the use of Xpath is seen in "for-each

select="studentlist/student..." part. This path locates the elements to be selected in the XML

document. It covers all the students in the "studentlist." Then, for every student the data between

the tags "fname," "lname," and "gpa" is extracted and mapped to the table created in increasing

order of gpa. The result of that transformation is shown in Figure 15.

F_Name	L_Name	GPA
Beth	Green	3.0
Joe	Brown	3.4
Carol	Red	3.7

Figure 15- Result of the transformation

Figure 15 illustrates the table containing student data, which is a result of applying

tezola.xsl in Figure 14 to tezola1.xml document in Figure 13.

2.5.4 XML Linking Language (XLINK)

In HTML it is easy to indicate hypertext linking, simply using the predefined "<A>"

element. Clicking on the content between the tags <A> and </A> allows one to traverse to

another document. The problem with XML is that it has no predefined tags. So how can a

hypertext link be represented in an XML document? A mechanism is needed to allow hypertext

link elements to be represented in XML documents, without giving up any of the advantages

gained by XML Specification 1.0, such as the independence from predefined tags.

The answer for this mechanism is XLL, the eXtensible Linking Language. Formerly

known as Xlink, XML Linking Language, XLL is a work in progress of the World Wide Web

Consortium (W3C) closely related to the XML Recommendation, but adds functionality for high-

function hypertext and hypermedia [34].

XLL allows hypertext link elements to be inserted into XML documents without using predefined tag names. To achieve that, XLL uses the reserved attribute "**xml:link**" to let the XML processor know that this is a hypertext link element and to take necessary action. There should follow some other reserved attributes which help to define a complete link element. For example, **HREF** defines the URL to the resource, and **SHOW** defines whether the resource document should be embedded in the current document or replace the current document or be displayed in a new window. An XML document having a combination of these attributes is shown in Figure 16.

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="try.xsl"?>
<!DOCTYPE documentlist [
 <!ELEMENT documentlist (document+)>
 <!ELEMENT document (title, abstract, word+)>
 <!ELEMENT title (#PCDATA)>
 <!ELEMENT abstract (#PCDATA)>
 <!ELEMENT word (#PCDATA)>
 <!ELEMENT simplelink ANY>
 <!ATTLIST simplelink
    xml:link CDATA #FIXED "simple"
    inline (true|false) "true"
    href CDATA #REQUIRED>
]>

<documentlist>
  <document>
    <title>Lowering the bar of dom api</title>
    <abstract>It talks a lot about
      <simplelink href="tezola1.xml">DOM</simplelink>
      and Java
    </abstract>
    <word>large</word>
  </document>
</documentlist>
```

Figure 16- Sample XML document containing hypertext link.

In Figure 16, a link element "simplelink" and its attribute list are defined in the DTD of the document. When the processor reads the "xml:link" reserved attribute in the "simplelink" element, it understands that it is reading values for a hypertext link. The link used in this example is of type "simple" (further explanation is given for the type of the links in the following

sections). In the XML data part, "simplelink" element for the "DOM" word is inserted into the contents of the element. This allows the user who views that XML document to click on the word "DOM" and to traverse to the "tezola1.xml" document which is assumed to have detailed information on DOM.

A brief elaboration follows on the types of links. The value for the "xml:link" attribute defines the type of the links; it can either be simple or extended. An analog to entity relationships can be drawn on to differentiate between these two kinds of link types. Simple link is a one to one relation, whereas an extended link is a one to many relation. From a link of type "simple" it is possible to access only one document, but from a link of type "extended", numerous accessible documents may exist. This is clearly depicted in Figure 17.

In Figure 17, at the top a sub-figure for simple link is shown. It is assumed that the content of the "abstract" element of the XML document in Figure 16 is styled with a style sheet and presented as in the sub-figure. The word "DOM" is the content of a simple link element, so when one clicks on the word "DOM," he is able to traverse to "tezola1.xml" document. The sub-figure at the bottom presents a general concept of extended links. It gives the ability to access many points from a single point. The "DOM_LINK" element contains three link elements. And when a user clicks on one of the words in the contents of the "DOM_LINK" element, he can traverse to all three documents automatically, or a popup menu showing all three links may appear and the user can choose the link to traverse.
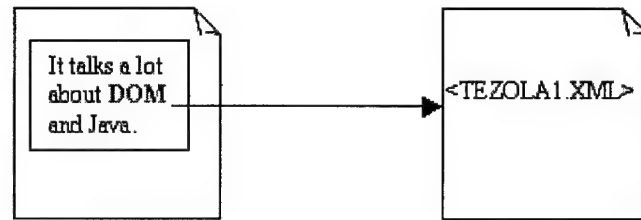
Simple Link
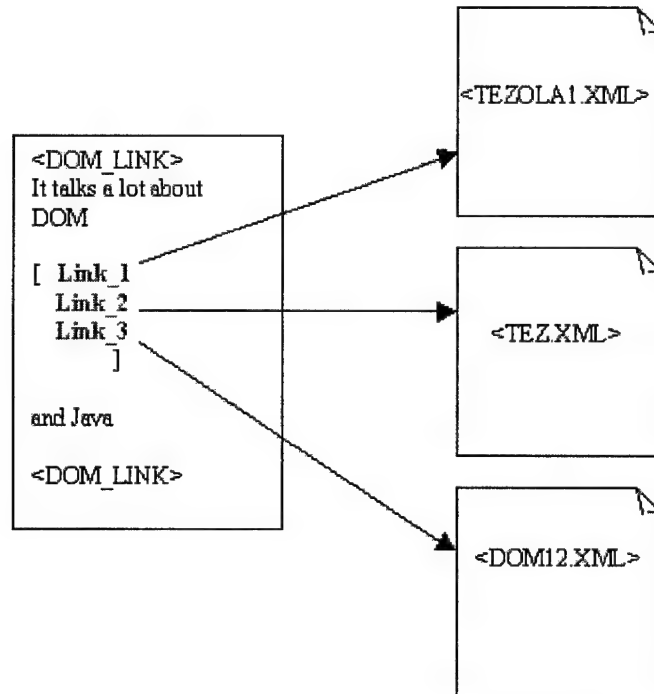
Extended Link

Figure 17- Simple and Extended Links.

Xlink gives the ability to traverse to a document, but what if the document is too big and the user wants to traverse to a specific part of the document. XLL can not provide this alone, but a combination of XLL with Xpointer produces the result wanted. A brief elaboration on Xpointer follows.

2.5.5 XPointer

Xpointer supports addressing into the internal structure of XML documents. It allows examination of a document's hierarchical structure and choice of its internal parts based on various properties such as element types, attribute values, character content and relative position. In particular, it provides for specific reference to elements, character strings and other parts of XML documents whether or not they have an explicit ID attribute [40].

As can be understood from the above description, Xpointer allows a browser to traverse an XML document to specific locations based on the element structure. In Figure 18, a fragment of an XML document which depicts the use of the combination of XLL and Xpointer.

```
<Weapon_System_Data>
  Role = "Technical Data"
  XML-Link = "LOCATOR"
  Href = www.wpr.com/weapons.xml#CHILD(2, Capability)
</Weapon_System_Data>
```

Figure 18-Combination of XLL and Xpointer.

In the figure above, "Href =www.wpr.com/wepons.xml#CHILD(2,Capability)" defines the fragment of the document to be traversed. "Href =www.wpr.com/wepons.xml" is what XLL is capable of and "#CHILD(2,Capability)" is what Xpointer provides. It tells the browser to traverse to the "weapons.xml" document then find the second child of the Capability element and return it to the user. As mentioned before, Xpointer makes use of the hierarchical structure in XML documents and consequently the keywords used in Xpointer like *Root, Descendant, Preceding, PSibling* are a reflection of this logic.

2.6 XML AND JAVA

XML allows a document writer to mark up data in a document in a platform- and language-independent manner, in other words, make the data portable across different platforms. The Java platform offers a hardware independent environment by offering a distributed, homogenous computing architecture with portable code that can be downloaded over a network to any Java virtual machine [37]. From a logical point of view, the match between Java and XML technologies is not surprising. Java was designed to develop applications on the Internet by providing independent programming environment and XML was designed to enable data to be exchanged or shared on the Internet by representing data in a common format. Thus, it can be clearly understood that Java and XML need each other to maximize their capabilities.

Both XML and Java make use of hierarchical structures. XML data is represented in a tree-like structure, and Java objects of an object model are structured in a tree-like structure too. So that makes the conversions from XML data to Java object and from Java object to XML data possible. In [1] it is stated that "The synergy between XML documents and Java code is due to the fact that the data encapsulated by XML tags can be easily expressed as Java objects and vice versa." Figure 19 illustrates this logic:
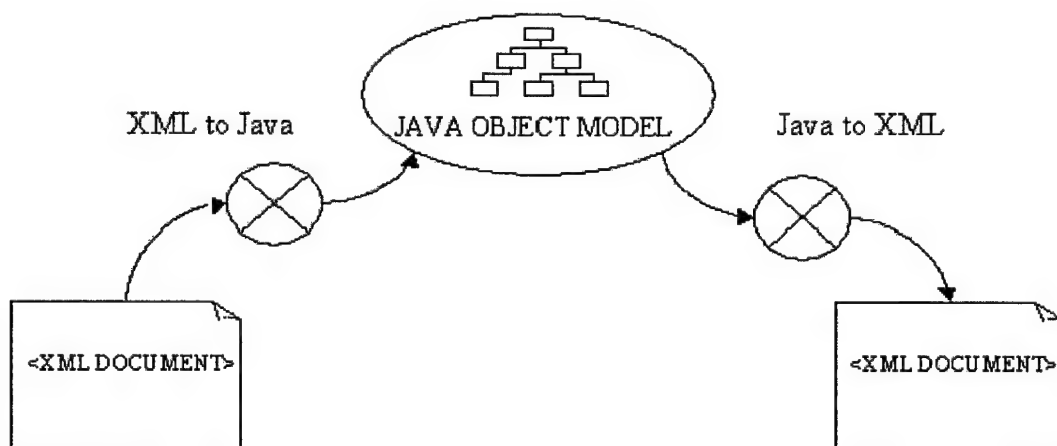
Figure 19- Conversions between XML and Java

As shown in Figure 19, conversions between XML and Java involve the following steps:

1- The structure of the XML document should be examined very closely. Strictly obeying the nesting of the elements and the relations between the elements of the document, a corresponding Java object model is created.

2- The Java object model is populated with data from the XML document. As a result the populated Java object model can be used to process the data wrapped in the objects.

3- The data stored in objects is written out to an XML file. Two advanced Java capabilities play an important role in achieving this goal: Java Reflection and Java Serialization. A brief elaboration on Java Serialization and Java Reflection follows.

Serialization is the process of converting in-memory representation of an object into stream of bytes that allows it to be written out to a file. The object written out is stored persistently. When needed, the original object can be created again by reading the object data kept in the serialized file.

The Java Reflection API allows the application developer to examine the structure of a Java object [3]. Constructors, interfaces, methods, and fields (attributes) of a Java object are accessible through this mechanism. The only constraint is that "private" field values are not accessible through this mechanism. This is a restriction of Java Security policies.

Writing out the state of an object in XML format makes the object data persistent. In addition, since the object data is represented in a common format, this data can be shared and used by other applications. For example, an application program written in C++ can read this data in.

2.7 OTHER USES OF XML FOR INTERMEDIATE DATA REPRESENTATION

"Intermediate data representations are a vital technique for facilitating the interchange of data from one platform/software package to another [4]." Figure 20 illustrates this idea.
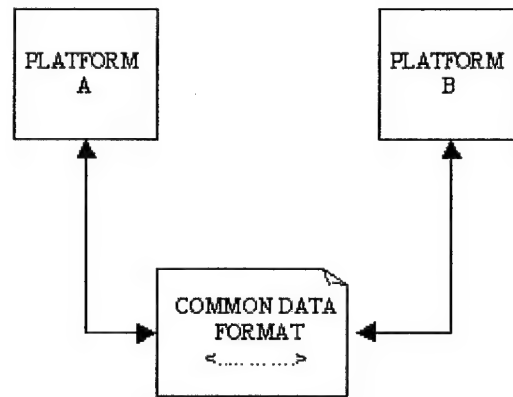
Figure 20- Intermediate data representation between different platforms.

In the figure above, Platform A and Platform B are different platforms which do not speak the same language. So how can the data in Platform A be transferred to Platform B or vice versa? The data needs to be converted to a common data format that both parties can understand. Once this conversion is done, data exchange between these two different applications can be done dynamically and accurately.

XML has this power; it is language and platform neutral. This issue becomes more important as the computer technology moves more towards distributed systems. Instead of centralized processing wherein one computer does every job involved in the systems, the current trend is to divide a whole system into atomic components and make one computer responsible for one component. When computer x needs data that can be processed by computer y, it can ask for the data, sending messages over the network to computer y. Computer y can prepare the data and send it back to computer x. This works fine as long as they both reside on the same platform. What if they reside on different platforms? Then there should be a protocol for determining the format for exchanging data, otherwise most probably the content of data will be misunderstood or misinterpreted.

Many examples of that are possible in business world. Imagine a company having two systems for its budget management-- incomes system and expenses system. For most of the time these two systems may work independently but they inevitably need each other. There should be a way set up to channel data from expenses system to incomes system or vice versa. This is widely known as "integrating internal applications." Enterprise application integration (EAI) can be thought in the same manner. [3]

2.7.1 XML and Databases

Data stored in databases is sharable, applying the predefined rules to extract data from databases (like SQL) any application can connect to databases and have an interaction with it. When exchanging data, what is mostly done is to exchange data between databases. Another good point of XML is that it is compatible with databases. A brief summary of "XML and Relational Databases" and "XML and Object Databases" follows.

XML and Relational Databases: An XML document's hierarchical structure can be converted into tables and the tables can be populated with data of the document. But when the structure of the XML documents is complex, RDB is not a good place to store XML data. The problem with RDB is mostly caused by the need to have foreign keys in tables to keep the relations between tables. When the XML document structure is too complex, it may be hard to create tables to reflect the structure of the document. So it can be concluded that for now it is best to use RDB when the document structure is not too complicated.

XML and Object Databases: As discussed before, XML can be parsed into object structures and the object structures can be stored in object databases. Object Databases (OD) do not have the problem RDB has. The complexity of the XML document is not an obstacle to storing XML data in OD.

A list of the areas where XML data representation may play an important role on consistent data exchange between different platforms and data sharing is given below.

1- Legal publishing

2- Collaborative CAD/CAM efforts

3- Collaborative calendar management across different system

4- Any corporate network application that works across databases, especially

   where policies must be enforced: purchase orders, expense requests, etc.

5- Exchange of information between players in any broker organized business:

   insurance, securities, banking, etc. [31]

2.8 SUMMARY

In this chapter, XML technology which provides portable data by representing data in a

common format and its related technologies like DOM, SAX, XLL, Xpointer, XSLT have been

discussed. Also a brief information on Java and XML compatibility has been given and the

outcomes of possible combinations of Java and XML technologies have been introduced. In the

next chapter, the theoretical basis of how these technologies can be implemented for this research

is discussed.

3 METHODOLOGY

3.1 INTRODUCTION

So far, the research topic, background for the problem related with research topic and the literature review has been discussed. In this chapter, the intent is to demonstrate methodologically, how to solve the problem and tighten the scope of the area of this work. In some parts of this chapter, although issues related to implementation are subjects of Chapter 4, some topics about implementation are introduced as necessary. Figure 21 is a synopsis of the research focus, showing basic concepts.
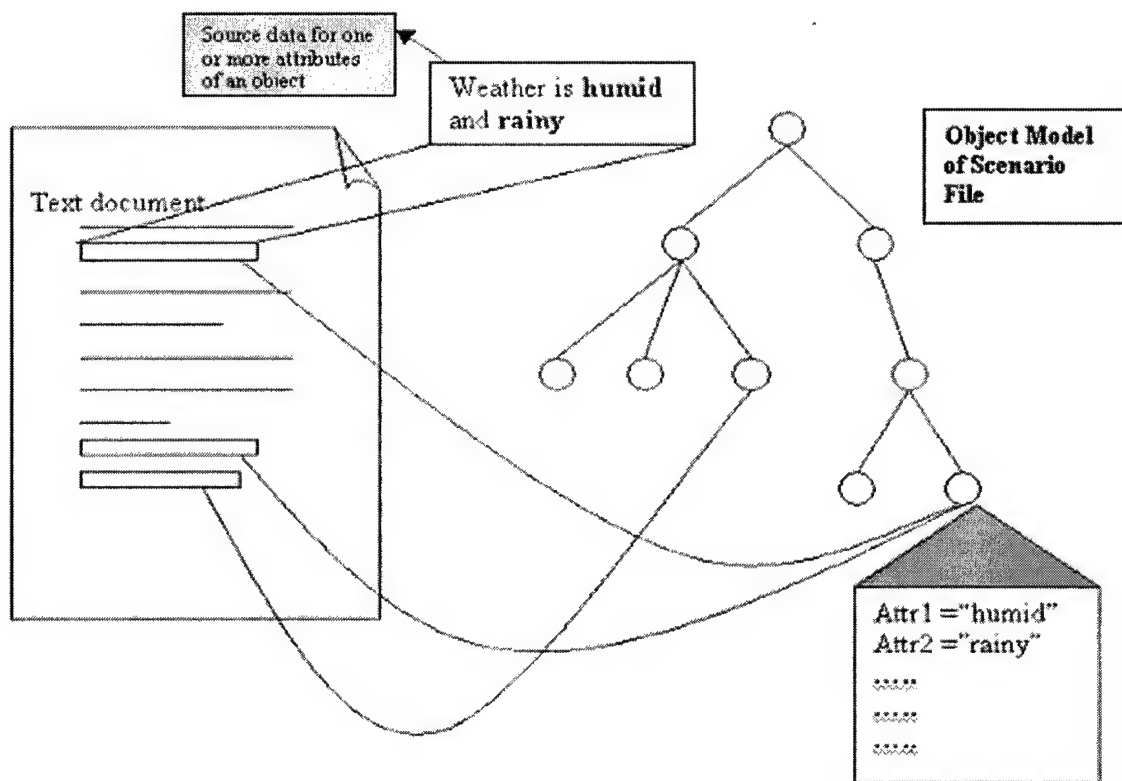
Figure 21- Scenario object source data.

In the figure above, there is a text document on the left hand-side which represents one type of ground truth data for one or more parameters of a simulation. At this point, it is important that the text document should have a well-defined structure, such as, at a minimum, paragraphs and sentences, as opposed to being simply data files or image files.

On the right hand-side, there is an object model, which represents a simulation scenario file, whose attributes are based on data from various parts of the text document, as well as other ground truth data source types. As is shown, there is at least a partial mapping between components of the document and attributes of the instances in the object model.

The objective is then, by making use of that mapping, to be able to trace back to the origin of data represented in the attributes of instances in the object model. This allows a scenario builder to specifically determine the attribute value's source location in the text document. Once this mapping is achieved, the user is able to view the origin of data dynamically, and the user can also update the values of attributes in the object model in the process of producing a new scenario file.

Here we also demonstrate the concept of scenario component reusability which is clearly beneficial, because it allows a user to make use of existing scenario files, modify the entries or values as required, then present it as a new scenario file. There is no need to create the whole scenario file from scratch again (as discussed in [18], the entry of data to scenario files is done by an analyst manually). The topics discussed in the present work make this process more automated and dynamic.

The most frequently used words in the preceding chapters are "XML," "Java," "Java DOM API," and "Xlink." It follows, then, that the solution for the problem should involve these concepts and the subjects related to them. Indeed it does, as the following sections show.

3.2 REPRESENTATIVE TEXT DOCUMENT

To ensure the classification of this document remains "unclassified," in lieu of authentic source ground truth data-bearing text segments, illustrative synthetic text with similar document structure is used. Actually the content of the document is not going to be so important. The content, made up for this research, is about general information about the flight characteristics of an aircraft. The data source mapping mechanism will be the same.

The most important thing about preparing this kind of a document is going to be representing data in a structured way in the document. Thus it will be very convenient for further manipulation, such as accessing specific parts of this document through applications.

3.3 CONVERTING TEXT DOCUMENTS TO XML

The source data of interest in this work is in the form of one or more text documents. It is just plain text so it is not self-describing. When data is in plain text format, it is going to be rather hard to perform dynamic information lookup and extraction in an application. The best way to manipulate and extract data from a text document is to "mark up" the text document using any of several popular markup standards, such as SGML, HTML, or XML; XML is used here. By doing that, an application can read and parse the document and build an index of tags, facilitating later access to specific parts of it.

As discussed in Chapter 2, XML (eXtensible Markup Language) is the most appropriate markup language to be used because of its simplicity and uncomplicated structure. One of the important properties of XML is tagging the meaning (or semantics) rather than the look (or syntax) of data, so a program can recognize the document easily. A text document can be converted into an XML document in three ways as discussed in the following sections.

3.3.1 Manual Conversion

Any "plain text" editor can be used for this purpose. The first thing to be done is to define the structure of the document. A Document Type Definition (DTD) should be declared before beginning to tag the document. A DTD is to an XML document what a schema is to database – it is metadata which describes the structure within the document.

DTD can also be used to "validate" XML documents. That is, the tag layout of the XML document is checked against the grammar described by DTD. If the structure in the XML document does not match the nesting, rules, and syntax defined in the DTD, the parser used is

going to give errors. Most parsers available require a DTD, like Sun's XML parser for Java (JAXP 1.0). There are also commercial parsers available which do not require a DTD. Details of the DTD and the significance of "well-formed" documents are described in section 2.3.1.

To achieve the goals set for this research, it is much more beneficial to create DTDs for XML documents and to use available XML parsers for Java which check against the DTD. That is the approach taken in this work.

The topics discussed above are the basic steps for creating an XML document. There are also some intermediate steps to be taken, but in this work the goal is not to introduce every detail of XML, but to explore the parts of XML related to the present work. To begin with, tools for XML manipulation are discussed next.

3.3.2 Using eXcelon™ Tools

The commercial product, eXcelon™, is a dynamic application platform which supports XML. The figure below, shows the eXcelon™ tool suite used with XML. [43]
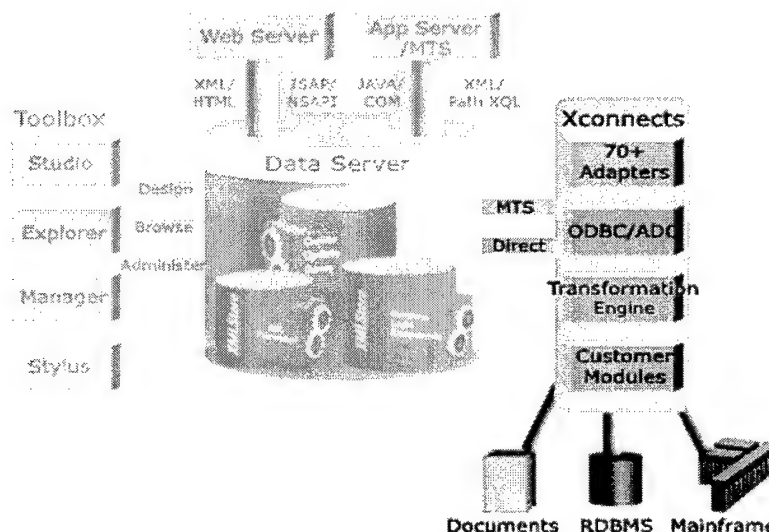
Figure 22- General view of eXcelon™.

The tools on the left side of the Figure 22, Studio and Explorer are the most useful tools to create XML documents. eXcelon™ Studio allows one to make a schema or define the structure of the

document as in the figure below. The schema depicted in Figure 23 shows the components of a document object, including a document title, abstract, and section headings. Notice that paragraphs are sub-components of a section, and sentences and words further define a paragraph.
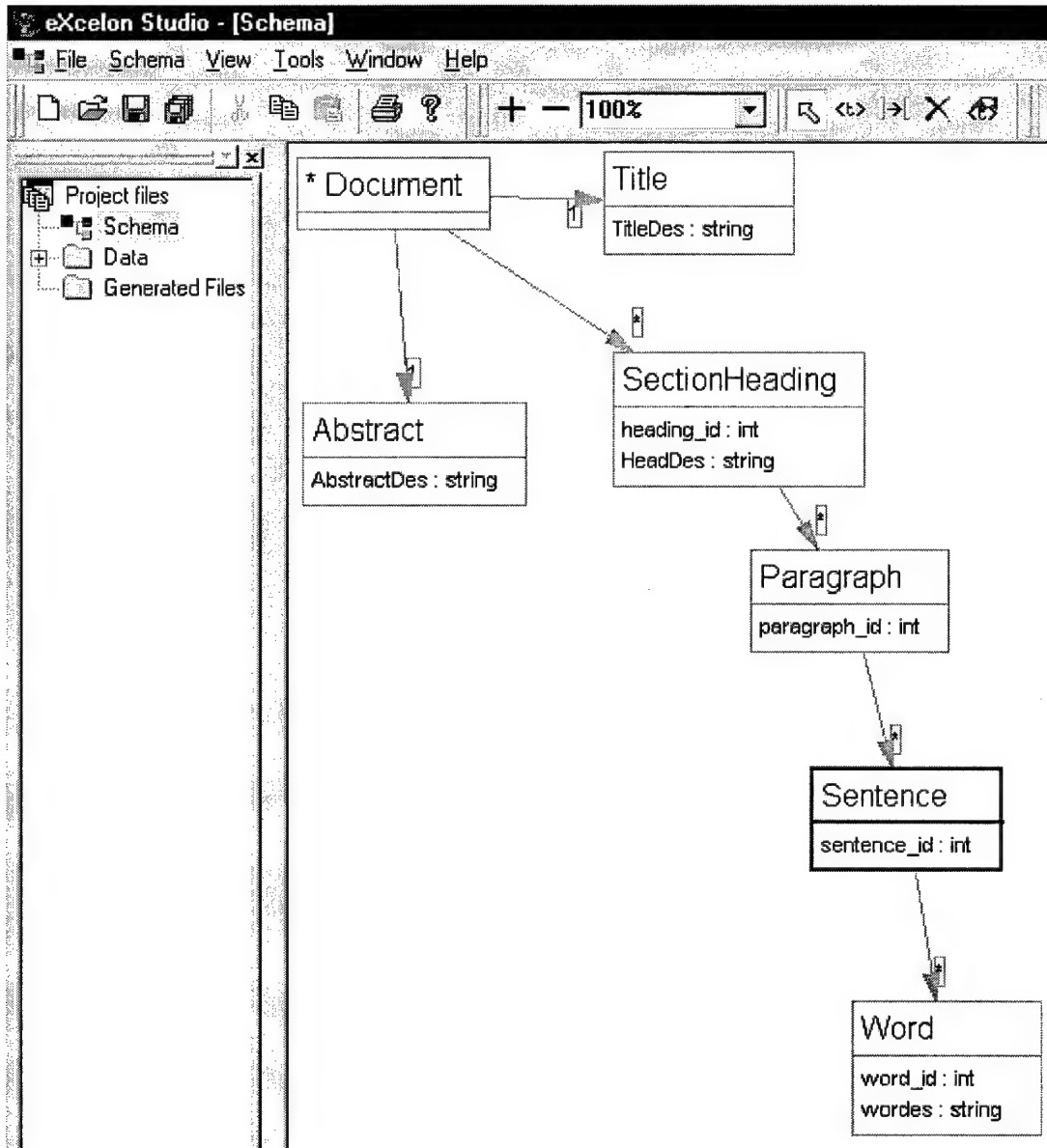
Figure 23- Defining the schema of the XML document using eXcelon™ Studio.

36

In accordance with the schema, eXcelon™ Studio will allow one to have a tree-like structure of the document, and this tree-like structure can be populated with data obeying the cardinality and nesting rules defined in the schema, as in Figure 24. For example, the figure depicts a single title and abstract, but many words inside of many paragraphs.
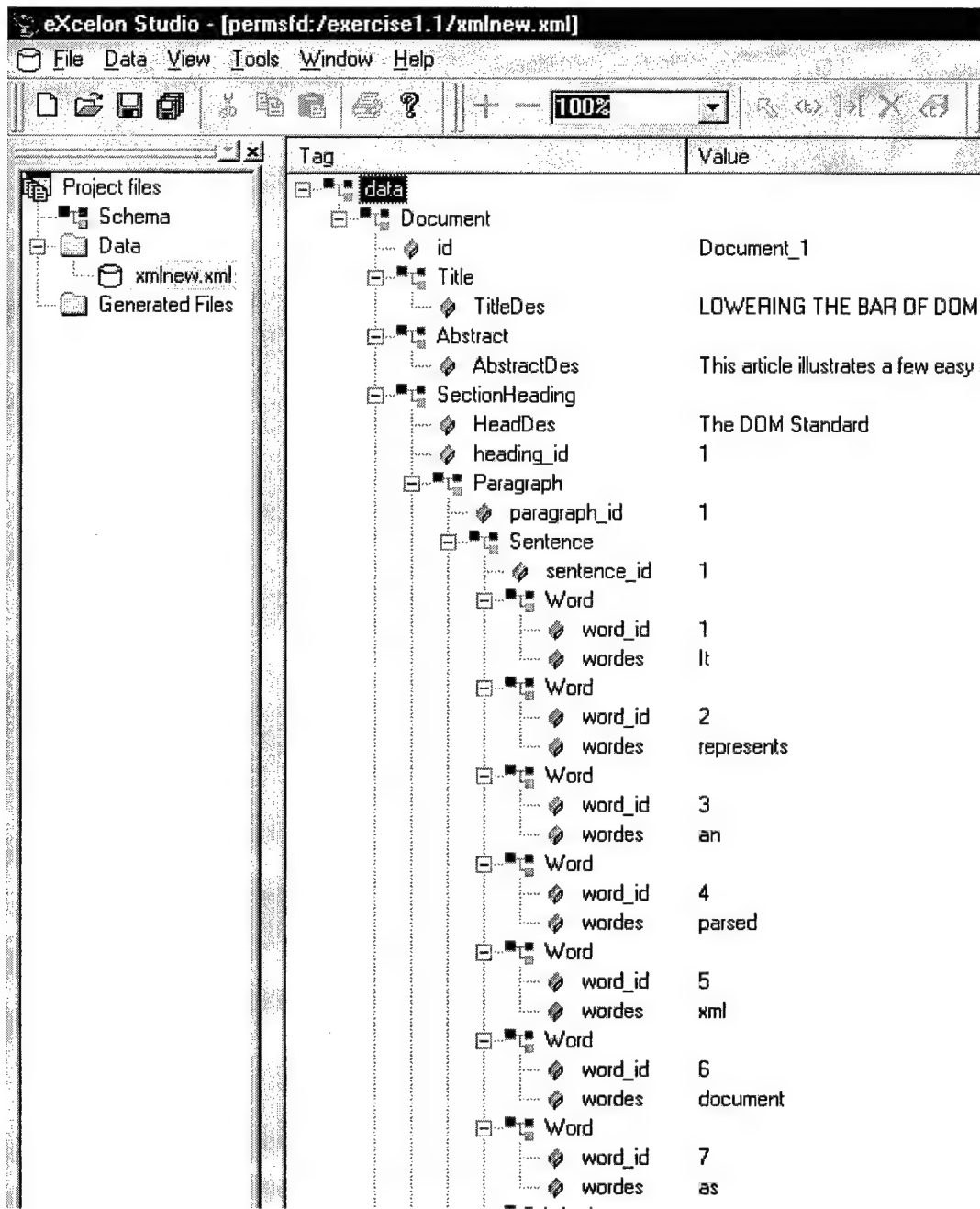
Figure 24- Populating the XML document structure using eXcelon™ Studio.

By using eXcelon™ Explorer it is possible to view the actual XML document created as depicted

in Figure 25.


```
eXcelon Explorer - [ennt05lj]
  Connection  File  Edit  Tools  Window  Help

  ennt05lj           <?xml version="1.0" encoding="UTF-8"?>
    bulk-load          <data>
    cars                 <Document id="Document_1">
    cars_mult              <Title TitleDes="LOWERING THE BAR OF DOM API"
    msfd-store             <Abstract AbstractDes="This article illustrat
    permsfd                <SectionHeading HeadDes="The DOM Standard" he
      exerc                  <Paragraph paragraph_id="1">
        al                     <Sentence sentence_id="1">
        e:                       <Word word_id="1" wordes="It"/>
        e:                       <Word word_id="2" wordes="represents"/>
        tit                      <Word word_id="3" wordes="an"/>
        xr                       <Word word_id="4" wordes="parsed"/>
    products                     <Word word_id="5" wordes="xml"/>
    teoman                       <Word word_id="6" wordes="document"/>
    videoctr                     <Word word_id="7" wordes="as"/>
                                 <Word word_id="8" wordes="an"/>
                                 <Word word_id="9" wordes="abstract"/>
                                 <Word word_id="10" wordes="tree"/>
                               </Sentence>
                               <Sentence sentence_id="2">
                                 <Word word_id="1" wordes="Only"/>
                                 <Word word_id="2" wordes="the"/>
                                 <Word word_id="3" wordes="interfaces"/>
                                 <Word word_id="4" wordes="reflect"/>
                                 <Word word_id="5" wordes="a"/>
                                 <Word word_id="6" wordes="tree"/>
                                 <Word word_id="7" wordes="structure"/>
                               </Sentence>
                             </Paragraph>
                             <Paragraph paragraph_id="2">
                               <Sentence sentence_id="1">
                                 <Word word_id="1" wordes="It"/>
                                 <Word word_id="2" wordes="is"/>
                                 <Word word_id="3" wordes="availbla"/>
                                 <Word word_id="4" wordes="from"/>
                                 <Word word_id="5" wordes="many"/>
```

Figure 25- The XML document viewed using eXcelon™ Explorer.

38

The only problem with this document is that it does not have a DTD. eXcelon™ has a tool, **XML Authority 1.1**, which can help view the DTD of the document created. When an XML document is imported to XML Authority 1.1, the result in Figure 26 is obtained. This illustrates the actual contents of the instance we created in Figure 24. The actual DTD appears in Figure 27.

Element Type	Text	Elem.	Content Model	Attributes
data	☐	☑	(Document)	
Document	☐	☑	(Title , Abstract , SectionHeading+)	id
Title	☐	☑	()	TitleDes
Abstract	☐	☑	()	AbstractDes
SectionHeading	☐	☑	(Paragraph+)	HeadDes, heading_id
Paragraph	☐	☑	(Sentence+)	paragraph_id
Sentence	☐	☑	(Word+)	sentence_id
Word	☐	☑	()	wordes, word_id
	☐	☐		
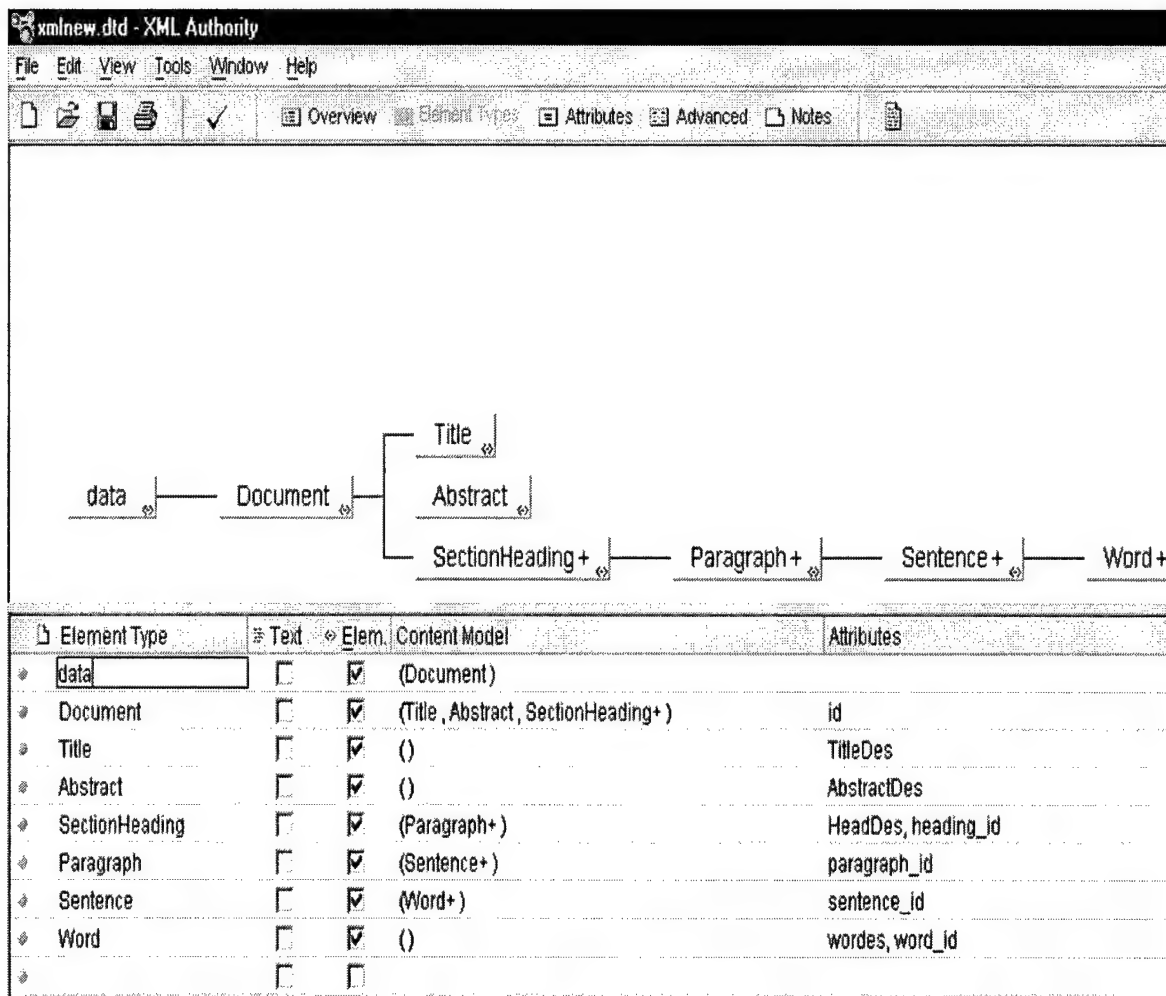
Figure 26- The XML document structure created by XML Authority 1.1.

39

When the menu selection view/source is clicked, it displays the DTD required by the structure

defined using eXcelon™ Studio.


```
xmlnew.dtd - XML Authority
File   Edit   View   Tools   Window   Help

   [icons]          ✓         ▣ Overview   ◁▷ Element Types   ▤ Attribute

Generated Source: C:\WINNT\Profiles\tyoruk\Desktop\xmlnew.dtd

<!ELEMENT data   (Document )>

<!ELEMENT Document   (Title , Abstract , SectionHeading+ )>
<!ATTLIST Document   id CDATA   #REQUIRED >
<!ELEMENT Title   ( )>
<!ATTLIST Title   TitleDes CDATA   #REQUIRED >
<!ELEMENT Abstract   ( )>
<!ATTLIST Abstract   AbstractDes CDATA   #REQUIRED >
<!ELEMENT SectionHeading   (Paragraph+ )>
<!ATTLIST SectionHeading   HeadDes     CDATA   #REQUIRED
                           heading_id CDATA   #REQUIRED >
<!ELEMENT Paragraph   (Sentence+ )>
<!ATTLIST Paragraph   paragraph_id  (2 | 1 )   #REQUIRED >
<!ELEMENT Sentence   (Word+ )>
<!ATTLIST Sentence   sentence_id   (2 | 1 )   #REQUIRED >
<!ELEMENT Word   ( )>
<!ATTLIST Word   wordes   CDATA   #REQUIRED
                 word_id CDATA   #REQUIRED >
```

Figure 27- The actual DTD.

It is possible to copy/paste this DTD to the beginning of an XML document (internal DTD) or

make a URL reference to this file in the XML document before the actual tagging starts (external

DTD).

3.3.3 Using Java DOM API

The third and final approach to obtaining a marked-up source document is by using the

Java Document Object Model API. By using Java DOM API, it is possible to build XML

documents from scratch. Normally, DOM is known to be used for representing an existing XML

document as a tree structure. But, Java DOM API also allows one to create an empty XML

40

document and then to add new elements to the XML document created, as needed. In the example

Java code in Figure 28, the basic operations for this purpose are shown.

```
import java.io.*;
import com.sun.xml.tree.*;        }  ────────▶   Required classes for the
import org.w3c.dom.*;                            DOM API

public class Sample
{
  public static void main (String argc [])
              throws IOException, DOMException
  {
        XmlDocument xmlDoc = new XmlDocument();
        ElementNode  country = (ElementNode) xmlDoc.createElement("country");
        ElementNode capitol = (ElementNode) xmlDoc.createElement("capitol");
        ElementNode population = (ElementNode) xmlDoc.createElement("population");

    Writer out = new OutputStreamWriter(System.out);

        out.write ("No tree exists yet\n");
        out.write ("\n");


        xmlDoc.appendChild(country);
        country.appendChild(capitol);
        capitol.appendChild(xmlDoc.createTextNode("\n Ankara \n"));
        country.appendChild(population);
        population.appendChild(xmlDoc.createTextNode("\n 70 million
\n"));

        country.setAttribute("name", "Turkiye");


    out.write ("XML DOCUMENT created by DOM API is:\n");
    out.write ("================================\n");
        xmlDoc.write (out);
        out.write ("\n");
        out.flush ();

        System.exit (0);
  }
}
```

Figure 28- The source code to create XML document via DOM API.

41

In this code there are three main nodes: **country, capitol and population**. Country has a "name" attribute and this attribute is set to "Turkiye." The other nodes do not have attributes. The nodes to be used are declared at the beginning of the program, but at that moment there is no relationship among the nodes yet. The nesting between the nodes, or the hierarchy, is set in the highlighted part of the code of Figure 28.

Population and **capitol** nodes do not have any attributes but they have text nodes that contain the values for population and capitol. The text node values represent the data content between tags. The output of this code is depicted in Figure 29. Note, the use of the Java DOM API in this way ensures a well-formed XML document.

```
C:\WINNT\System32\CMD.exe

\ODI\osji\;C:\WINNT\system32;C:\WINNT;C:\WINNT\System32\WBEM;D:\JDK1.
IMNnq_NT;D:\MSSQL7\BINN;D:\ORAWIN\BIN;
PATHEXT=.COM;.EXE;.BAT;.CMD;.UBS;.UBE;.JS;.JSE;.WSF;.WSH
PROCESSOR_ARCHITECTURE=x86
PROCESSOR_IDENTIFIER=x86 Family 6 Model 7 Stepping 3, GenuineIntel
PROCESSOR_LEVEL=6
PROCESSOR_REVISION=0703
PROMPT=$P$G
SMS_LOCAL_DIR=C:\WINNT
SMS_LOCAL_DIR_USER=C:\WINNT
SystemDrive=C:
SystemRoot=C:\WINNT
TEMP=C:\TEMP
TMP=C:\TEMP
USERDOMAIN=AFIT
USERNAME=tyoruk
USERPROFILE=C:\WINNT\Profiles\tyoruk
windir=C:\WINNT
No tree exists yet

XML DOCUMENT created by DOM API is:
====================================
<?xml version="1.0" encoding="Cp1252"?>

<country name="Turkiye">
  <capitol>
 Ankara
</capitol>
  <population>
 70 million
</population>
</country>
```

Figure 29- The simple XML document created by DOM API.

42

When the three methods proposed for creating XML documents are compared, the DOM API is the most complex to use, and the XML document created does not have a DTD. The sample XML document demonstrated above is a simple one. But as the text document structure gets more complex, the harder it is going to be to transform the text document to an XML document using DOM API. Using eXcelon™ is much easier, because it is generic. That does not mean that eXcelon™ is completely satisfying for creating any kind of XML document to any level of complexity. For the parts which eXcelon™ is not well suited, manual methods can be applied.

3.4 HOW TO TRAVERSE FROM OBJECT GRAPH TO THE DATA SOURCE

Owing to the work of McDonald there exists a collection of over 200 Java classes to define an object model for a Suppressor scenario file-set. In [18], a Multi-Spectral Force Deployment (MSFD) database file is analyzed as a prototypical data input source file used in scenario construction. The object graph of McDonald, populated by parsing the MSFD database file, is visually represented to the user by a Java tree in a GUI frame. The point where we take over from McDonald is shown in Figure 30.

The figure below demonstrates how a populated Suppressor object model is presented to the user in a hierarchical fashion. The application developed by McDonald does not have the capability to trace the origin of source data for the attribute values of the Suppressor object graph. This research takes over from McDonald's work at this point and adds this "source-data-traceability" capability to the existing work.

43

```
CERTCORT Suppressor Datagateway Agent                    _ □ ×
File    Help
                        @UNREGISTERED@
□ ⬜ SUPPRESSOR                                              ▲    Model
  ⬜ TDB                                                         Chains
    ⊞ ⬜ PLAYER STRUCTURE                                         Load
    ⊞ ⬜ CAPABILITY
    ⊞ ⬜ SUSCEPTIBILITY                                           View
    ⬜ TACTIC                                                     Agent
      ⬜ sam-a_tactics
        ⬜ ASG-CMD-CHAIN
          ◆ sam-a_chain
        ⬜ INTELL-REPORT-FREQ
          ⬜ RPT-RATE (1/SEC)
            ⬜ sam-a_chain
              ◆ 0.0333
            ⊞ ⬜ ew/gci_chain
            ⊞ ⬜ intell
        ⬜ MAX-MSG-ATTEMPTS            ┐  Sample of attribute values to be
          ◆ 4                         │  traced origin of source data,
        ⬜ MAX-SNR-PERCEPTIONS         ├  McDonald's application does not
          ◆ 7 (TGTS)                  │  have this capability
        ⬜ MOVE-TO-ENG                 ┘
          ◆ NO
        ⬜ CMD-CHAIN-TYPES(REPORT-RESPONSIBILITY)
          ⬜ intell
            ◆ CMDR
        ⬜ SALVO-FIRING
          ⬜ TGT-MOTION
            ⬜ APPROACHING ROUNDS-FIRED (ROUNDS)
              ◆ 3 2
            ⬜ RECEDING ROUNDS-FIRED (ROUNDS)
              ◆ 2 1 1
        ⊞ ⬜ SNR-RPT-GUIDE
        ⊞ ⬜ ZONE-CHARACTERISTICS
        ⊞ ⬜ RESOURCE-ALLOCATION
      ⊞ ⬜ sam-a_missile_tactics                              ▼
```

Figure 30- How McDonald visualizes his populated object graph.

In the following sections of this chapter, the mechanisms which allow one to traverse

back to the source of data for the attribute values of Suppressor object are introduced. In so doing,

we explore the capabilities of Java and XML.

3.4.1 Mirror-Tree Model

In this first approach, McDonald's JTree representation of the Suppressor object model is used as the basis. The frame McDonald used is split into two panes. The first pane has the original JTree representation of the Suppressor object model depicted in Figure 30. The mechanism of McDonald's application is shown in the figure below.
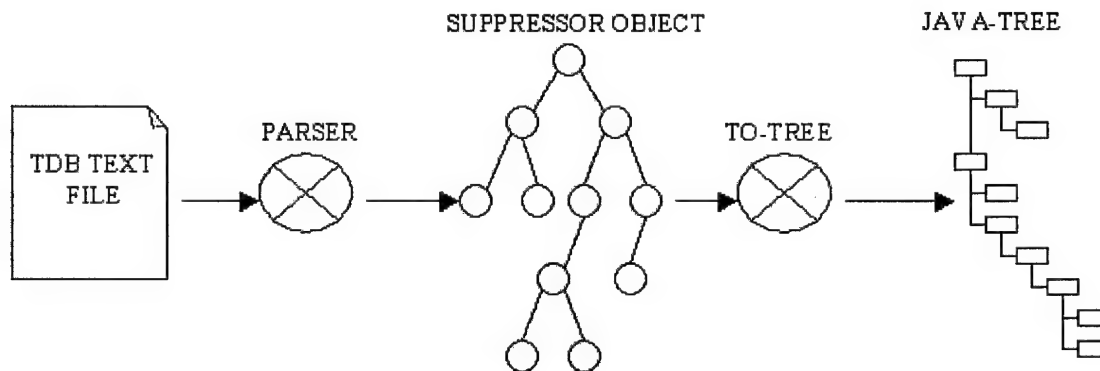
Figure 31- How TDB text file is parsed into object model and Java Tree.

As demonstrated in Figure 31, a TDB text file (flat file) is parsed into objects of "Suppressor" and then the data parsed into the object model is visually represented by Java tree. He has *toTree()* and *parse()* methods present in almost all of the objects. When an input-stream is read from the flat file, the tokens of that stream are fed into the mapping object models by calling the *parse()* method of that object. In every call to the object's *toTree()* method, new nodes are created in accordance with the data item parsed into the object read (sometimes a single node and sometimes a subtree is created in these methods). The nodes or the subtrees returned by *toTree()* method are added to the main tree. The population of the object model continues until all of the flat file is parsed into tokens.

Parsing the TDB file into a JTree is straightforward. The real work of this research is in tree object to source paragraph mapping. In Figure 32, the first attempt to trace source data in this research is demonstrated.

45
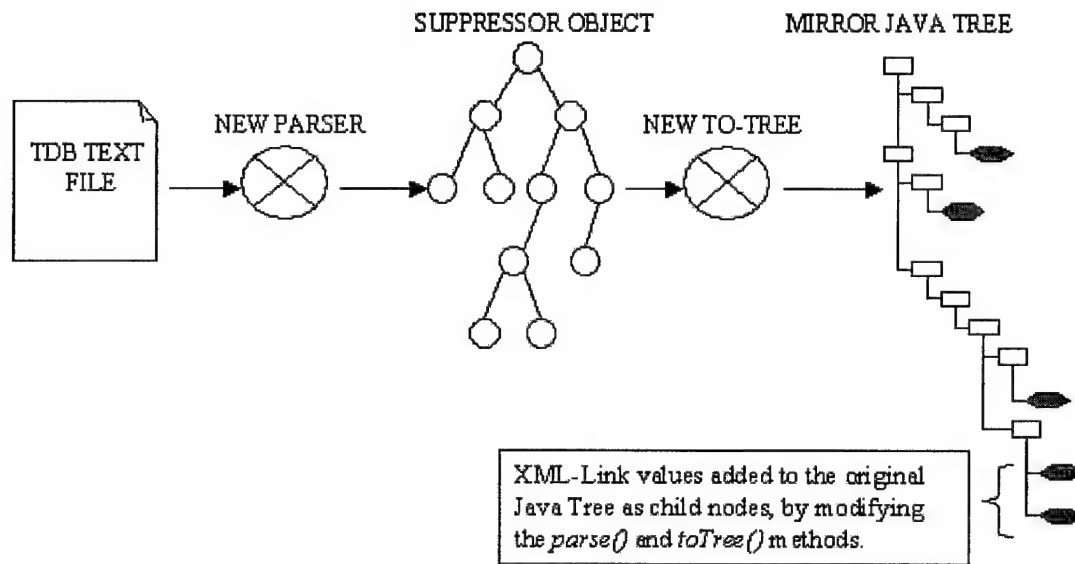
Figure 32- The new Java tree representation obtained as a result of modified parse() and toTree() methods.

In this approach, a new text file called "TDB_M.txt " ("M" for "mirror image") is created in order to store the links to the XML documents. This new file is almost the same as the original one, but the links are added where appropriate. As McDonald did, *toTree_M()* and *parse_M()* methods are added to all of the objects. *ToTree_M()* methods add the links which represent the attributes' values as children to the leaf nodes. Hexagonal-shaped child nodes on the new JTree representation in Figure 32 are the xml-link values added to the original JTree. The JTree which contains the link values is made visible on the right split-pane. So at this point there are two JTree representations; the original JTree and the mirror JTree which has the xml-link values. To find corresponding link values there is a need to traverse from original JTree to the mirror JTree as shown in Figure 33.

46
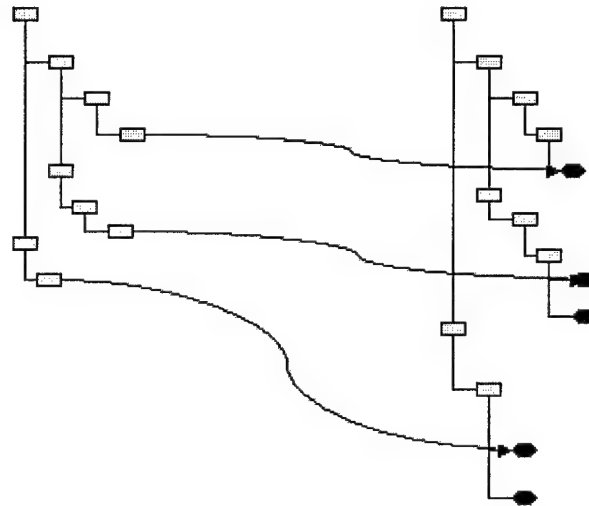
Original Jtree Representation   **New Jtree Representation**

Figure 33- Mapping between the original JTree representation and the mirror JTree.

Making the new JTree visible is not required to find links, because the new JTree "root"

or "handle" is already assigned to a variable which is accessible by the tracing application. All the

operations are done by calls to that variable, not by making use of the demonstration of the new

JTree. So it is optional to show the new JTree.

On the original JTree when a node is selected to trace to the origin of its source data, its

path to the root is stored in a variable and this path variable is passed to the method which finds

the links. The *findLink()* method follows the same path passed as the parameter and finds the

node. Since the links are added as the children of nodes, all the children of that node are the xml-

links required to find the source of the data. The XML parser is called iteratively as long as the

node has children. This is because an attribute may have more than one xml-link. The function of

XML parser is depicted in Figure 34.
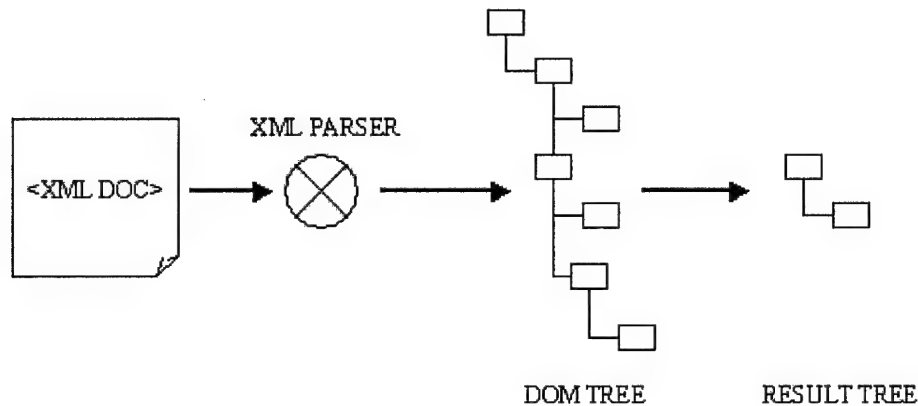
Figure 34- XML parser's function..

As shown in Figure 34, when the XML parser has the link, it first finds the XML document then reads in that XML document and builds a tree-like structure (DOM) of the XML document. By using the available methods in Java DOM API, it extracts the data from the tree-like structure ("Result Tree" in the above figure) and returns that data to the user in a display window in a tree-like fashion.

The user does not generally require the ability to edit the source data, but may be given the ability to edit the source of data if necessary. In this work, it is assumed that the user does not manipulate the original source data by this GUI mechanism. Another important point to mention about the first approach is this: in McDonald's work there is no capability to make changes on the JTree representation of the Suppressor. In a small scale, it is shown that it is possible to edit the values of the nodes on the JTree and save these changes to the object model. Once these changes are made, the object model can be serialized using Java's native serialization methods, and later on when needed, this object model can be read in from the serialized form. This allows the user to update the scenario file object model. Then the updated object model can be written out as new text file which is going to end up as a new scenario file for further manipulation.

The solution to write out the modified object model is to use the **"export"** function provided by McDonald. This function allows one to write out the object model as a new

syntactically correct Type Database (TDB) text file. When writing the object model as a new TDB file, McDonald uses a similar approach he used in populating the object model and building the tree. He has *phrase()* methods in most scenario-specific objects. In those methods the string patterns which are needed to mark the TDB file to make it parseable by Suppressor are hard coded. The markup and the current values in the object model are added to each other. Every object's *phrase()* method returns the stream, to which it is related and they are all appended to the new TDB text file. So the changes made are made persistent and this new TDB text file is given a name or a version number to distinguish it from the other TDB text files. In the future when somebody needs this specific TDB file, it can be visually represented as a JTree. This is an important point, because it allows the scenario builder to view the source of data. Further depending on the source of data to make updates on the JTree and reflect those changes to the object model and have a new TDB text file in Suppressor-readable form.

If the user need not see the new JTree (no visual representation is needed or for the cases the information stored in the object model is enough), then there is no need to use the export function. Using the Java serialization mechanism is sufficient for this purpose. Serialize "out" the object model and serialize it "in" when needed. This is a key technology inserted into this ongoing work at AFIT in support of the AFRL simulation integration effort.

3.4.2 Link Builder Model

The first approach has some drawbacks:

a. The XML links to attribute source data are added to each TDB file manually and it is tedious.

b. The person in charge of adding the links has to be very careful with the syntax used, because the rules defined to parse the flat file may cause some parsing errors or exceptions.

In this alternative approach, the effort is to make the preparation of the links more dynamic and less error prone and to find a new way to store the xml-link information. It is assumed that the scenario builder and the link builder are not necessarily the same person. The link builder is conceivably a process on a server in charge of building xml-links for a given JTree. The scenario builder is the client demanding xml-links for the scenario -in- work he has. The basic steps taken in this approach are demonstrated in Figure 35.

Scenario Builder(Client)   **Link Builder(Server)**

Network Connection

1. Client asks for the link values for Jtree representation of Suppressor by sending the Jtree object to the server.

2. Server prepares the xml-links for the specific Jtree and sends back the container which has the link objects to the client
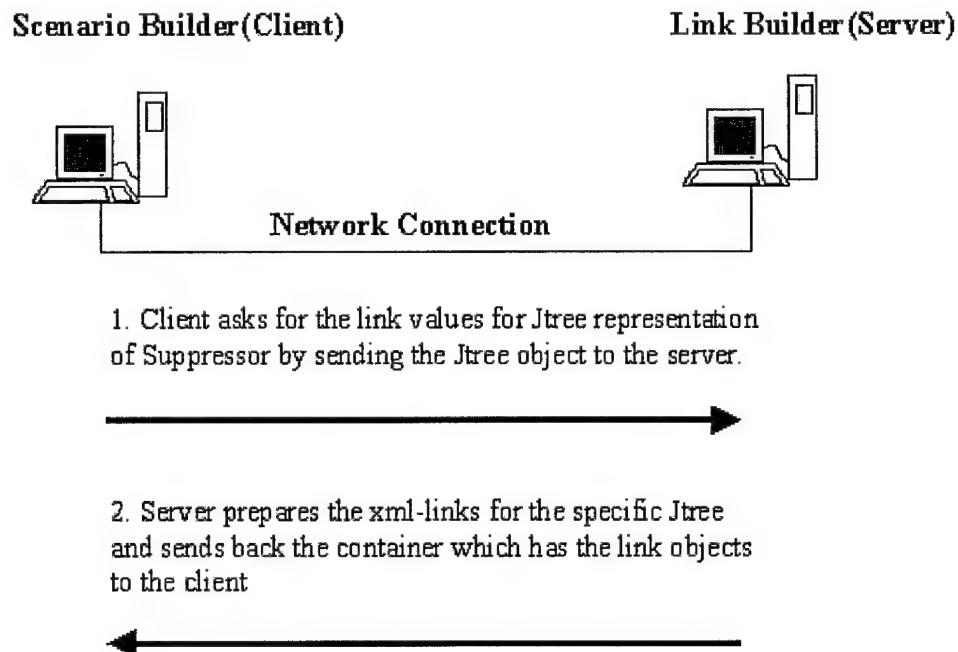
Figure 35- The communication between the scenario builder and the link builder.

It is possible to send objects over a network from one computer to another using output and input-streams provided that the objects to be sent over implement the Java Serializable interface. The side receiving the input-stream has to cast the stream as the desired object, otherwise it is not known which object this input-stream represents.

As shown in Figure 35, the scenario builder sends the JTree representation of the Suppressor to the server and asks for the links for the nodes in the tree. The server casts the sent object as a JTree and then makes it visible to the link-builder on a JFrame. When the link-builder

selects a node to add an xml-link, he is asked to enter the xml document name, tag name and attribute value via a "DialogPane." The path of the selected node to the root and the other link values provided by the link-builder are mapped into a link-object which has the attributes "path," "xml-document-name," "tag-name," and "attribute-value." Then this object is added to a container which has the ability to store the link objects. The link builder defines the links for all of the nodes in the JTree in the same way. When the link builder is finished with preparing the links, he sends the container object back to the client (the scenario builder) over the network.

After receiving the container object of the links, the client assigns the container object to a variable from which it can extract link data with get/set methods. The system is ready to be used to find xml-links. When a node is selected to find the source of data, the path to the root is assigned to a variable. And this variable is compared to the "path" attribute value of the link objects in the links-container. If they are equal, the link object's "xml-document-name", "tag-name" and "attribute-value" values are passed to the XML parser as link parameter. And the rest of the process is the same as discussed in the first approach. Again, the Java DOM API is used to extract data from XML document's tree-like representation.

3.4.3 Drawbacks of the First Two Approaches

In both of the demonstrated approaches, the path to the root of the selected node is the main parameter used to make the mapping between the data and the link value for the source of data. The drawback with these approaches is that the tree structures have to be the same on both sides (the original JTree which represents Suppressor object and the mirror JTree created to contain link values.) If the tree structures are not consistent there is possibly no corresponding link. Also the link results are too deterministic. That is to say if a link is found, it is assumed that it matches 100% and all the other links are irrelevant. (This situation is parallel to the problems associated with the Boolean Model of Information Retrieval). The system needs to be more flexible. Also the preparation of the links is not 100% dynamic, still somebody has to type in the

links, which consists of the XML document name, element (tag) name and the identifier attribute value of the element. In the two previous approaches discussed, the links are typed in either augmenting a TDB text file as is the case in mirror tree model or via a user interface as is the case in link builder model.

Another point of concern is, for both of the approaches, either SAX or DOM is used. We seek an alternative API to extract information from XML documents compliant with the XML 1.0 Specification. Further, we seek to address the applicability of OODBMS to this research. The proposed approaches' effort is to address the above issues, make the system more flexible and minimize the drawbacks discussed.

3.4.4 Meta-Class Instance Model

To be independent of JTree paths, a new mapping between the data and the links of the data source is needed. For this purpose a new object model is needed to reflect the structure of the original object model and to keep the link values. The exact name for this model is "meta-class instance model." For example, a Tactic object can have objects like "RcsTable," "TgtReflectivity," etc.. There is not a corresponding object for every instance of a Tactic's every instance of RcsTable object. All of the objects of type RcsTable of the original object model are mapped to only one instance of RcsTable in the meta-class object model. The meta-class instance model has only one instance for the corresponding instances in the original object model. A more clear view of that model is given in Figure 36.

An instance of Tactic Class   Another instance of Tactic Class
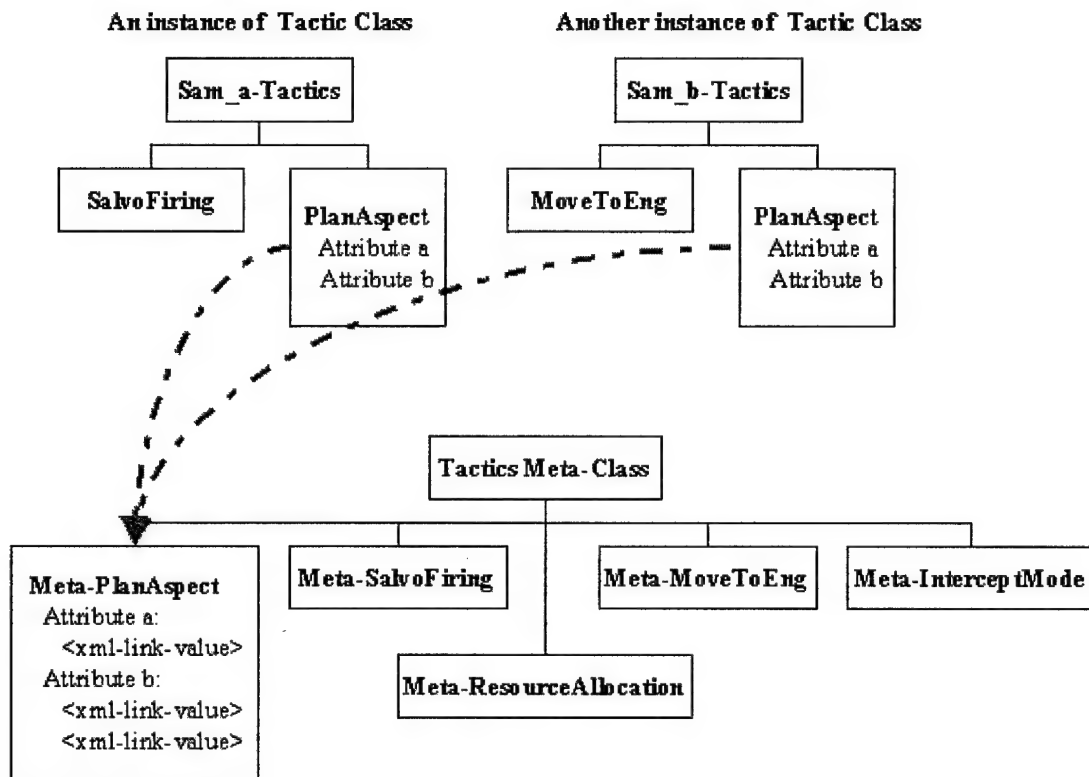
Figure 36- Mapping between object models.

A tactic class in the object model has almost 30 subclasses. But an instance of a Tactic class need not have all these subclasses. In Figure 36, Sam_a Tactics has the objects SalvoFiring and PlanAspect, and Sam_b Tactics has PlanAspect and MoveToEng objects. And the mapping is shown with the curved lines. The important point to mention about the figure is that there are two instances of PlanAspect class in Sam_a and Sam_b Tactics, and they are mapped to the *same* Meta-Plan Aspect instance in the Meta Class Model (there exists only one instance for Meta-PlanAspect and that is also valid for all the objects in the meta class instance model.)

The meta-class instance model can either store the actual links to the XML document or pointers to the links. For example, the link-objects introduced in Link Builder Model can be mapped to a Relational Database (RDB) table. Every object in the meta-class object model can be associated with an SQL query whose result-set returns the required xml-links from such a table. The parameters in the meta-class object model help create the SQL query statement generically.

3.4.4.1 XSLT Model

Until now, either a DOM or the SAX API is used to extract data from XML documents. The XML specification provides some other API's to extract information from XML documents. XSLT also has the ability to access data in XML documents. In fact, the first idea was to exploit XML Query Language (XQL), which has a query structure and syntax very similar to SQL, to extract information from XML documents. XQL is an adjunct language of XML which extends from XSL. But due to the fact that XQL is a standard which is still in development and not yet mature, the parent of this language, XSL, was chosen to be used in this work.

It is known that XSLT gives style to XML documents, but in addition to that it can be used to effect transforms on XML documents. That is to say, it can be used to extract data selectively. The transformation capability of XSL, which provides an alternative mechanism to access and manipulate XML documents, is our focus in this research. Figure 37 demonstrates how XML and XSL work together.
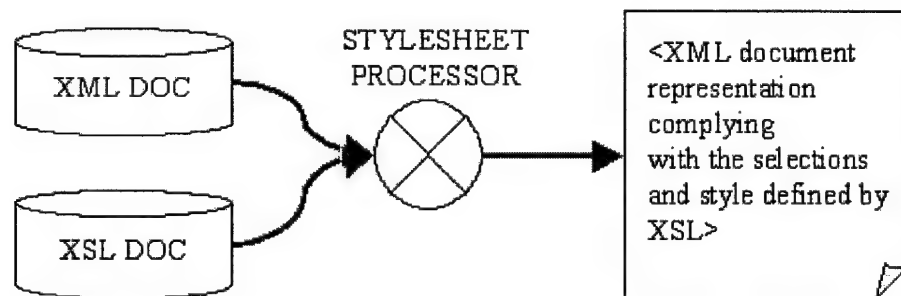
Figure 37- How XSLT works with XML.

To have the transformation shown in Figure 37, an XML document needs to be associated with a style sheet. That style sheet determines how to style the document and also which parts of the document should be made visible. The end product of the transformation is

again an XML document, but which conveys to the user a document stripped of the tags and having the data which the user is interested in.

The figure above gives some insight to how XSLT can be implemented in this research. There is a need to associate the objects in the meta-class instance model with a style sheet. The original method was to store the XML document name, tag name and attribute value as the link. The new method proposed is to replace these link values with values which serve to write out the XML style sheet generically. Those values are the XML document name, the Xpath to the specific element, and the attribute value of the specific element to be extracted. Consequently, the generically written XSL serves to fetch the related portions of an XML document.

At first glance this may seem like creating as many style sheets as the number of attribute values in the meta-class object model. By adding a method, *create_XSL()*, to a class in the object model in order to form a style sheet dynamically, there is no need to prepare numerous style sheet files manually. Also the required parameters for the style sheet are kept in the meta-class instance model. To make this point clear, consider the "Meta-PlanAspect" class which is a subclass of "TacticsMeta-Class" depicted in Figure 36; it is modified as in Figure below.

```
Meta-PlanAspect

Attribute a:
<xslt_info>
  XML-Doc-name,
  Xpath,
  Element's Attribute Value
</xslt_info>

Attribute b:
<xslt_info>
  XML-Doc-name,
  Xpath,
  Element's Attribute Value
</xslt_info>
```
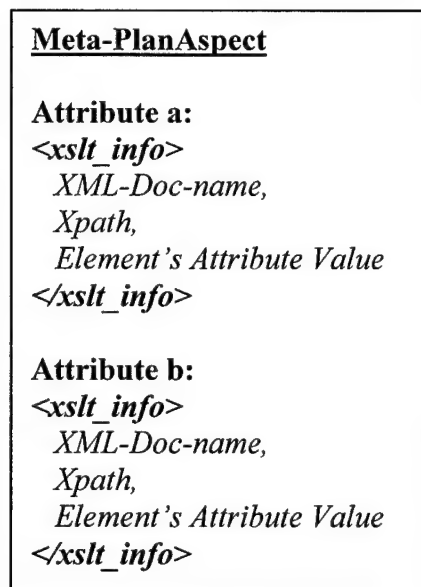
Figure 38- Modifications on the meta-class instance model to store XSLT values.

As depicted in Figure 38, the attributes of the Meta-Class Instance Model's objects have the required values to form the XSLT dynamically. When a user wants to find the data source for an attribute value in the Suppressor object model, the corresponding meta-object is found in the object graph of meta-classes. By making use of the "<xslt_info>" encapsulated in the corresponding meta class' attribute (which consists of the name of the XML document to be transformed, the Xpath value which helps to locate the specific element in the XML document, and the attribute value of the specific element), XSL is formed. Then another method takes over the job to invoke Internet Explorer 5.5 (I.E 5.5) to apply the "style" and the "transformation" defined by the XSL to the XML document specified. The final form of the XML document (data source) is returned to the user via the Web browser's window.

When compared to the previous methods which use the DOM, this approach allows one to view a data source as a small XML document which contains the original XML document's related parts, additionally it has style and has no tags. When the data source has a table-like structure, or the data-source is too complex and needs to be reorganized when presented to the user, or when the data source is a type other than a text, like an image, it is much more beneficial to use the XSLT model to trace the source of data and present that source of data to the user. There is no need to build and keep a tree structure (DOM) of the document in memory in such cases.

3.4.4.2 Editing the Meta-Class Instance Model's Link Values

It is important to be able to modify the link values of a meta-class instance model. In time, the source of data may change or there may arise a need to make the link value more precise. To make the system more flexible, the user is given the ability to the edit the link values depending on the results returned to him from the original source XML document. The current value of the link brings an XML document fragment as the data source, but the scenario builder may find the data returned irrelevant or too general. Upon reading the document fragment, the

scenario builder may decide that this document fragment is not the actual data source (irrelevant), or he may also decide that there is no need to view the entire document fragment (too general), and just to view a portion of the document fragment is enough. At this point, the scenario builder is given the capability to modify (make the link more specific or more general) or completely change the link value (define a new xml-link to replace the original xml-link).

Once the meta-class instance model's link attributes are modified, there is a need to preserve the state of the objects of the meta-class instance model between runs. This is achieved in two ways:

Serialization: By implementing the Java Serializable Interface, the modified meta-class instance model is written out to a file, and when the application is run later and the modified meta-class object model is required, it is read in from the file.

Serialization is effective and efficient, as long as the number of objects to be serialized is small, because serialization has to read and write the entire object graph at a time. Serialization also has drawbacks when there is a need to update objects frequently. Dynamic queries can not be processed against the files which have the serialized form of the object graphs. In addition, serialization does not provide reliable object storage. If the application crashes when the objects are being written out to a file by serialization, the contents of the file are lost.

ObjectStore Object Oriented Database Management System (OODBMS):

ObjectStore provides the mechanisms to overcome the disadvantages of serialization listed above. In fact, the steps to make objects persistent show some similarities to serialization. As with serialization, ObjectStore also makes use of **"readObject"** and **"writeObject"** methods, which are automatically generated for each persistence capable class. But the superiority of ObjectStore over serialization is that it has the reliability of database management system (DBMS). This provides reliable object management and integrity. In addition to that, accessing and manipulating the persistent objects have in-memory-like performance [45]. It has a better performance for accessing large number of objects. It is possible to extract information from the persistent object

graph by executing queries. Relational Database Management Systems (RDBMS) also provide

robust and reliable data storage, but there is overhead with mapping Java objects into relational

database tables and writing extra code to implement this mapping.

As a result, we conclude that it is much more efficient to store the entire meta-class

instance model in ObjectStore owing to the abilities provided by ObjectStore. This idea is

illustrated in Figure 39.

Java in-memory Meta-Class Instance Model

Meta-Class Instance Model stored in ObjectStore
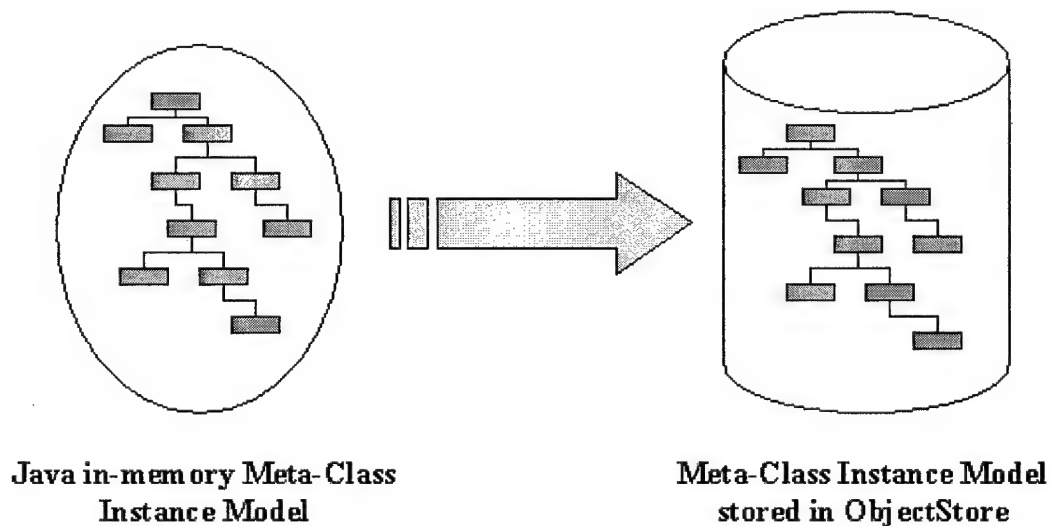
Figure 39- Storing Meta-Class Instance Model in Object Store.

As demonstrated in Figure 39, modified meta-class instance model's state is preserved

between runs by storing the whole meta-object graph in ObjectStore. When the scenario builder

needs that specific persistent meta-object graph, by the capabilities provided by ObjectStore, the

scenario builder can access and manipulate the meta-object graph as if it were in-memory meta-

object graph. He is also able to prepare queries and execute these queries on the object graph

saved in ObjectStore. This is an efficient way to modify, store and access the modified links. The

valuable xml-links created before are never lost.

3.5 INFORMATION RETRIEVAL ASPECT

This research also has an "information retrieval aspect." As a reminder, what is done in this work is to traverse from the attribute values of an object graph representation of a scenario to the source of data represented in XML format. Or, to put it in other words, we are attempting to retrieve information from XML documents, taking the values provided by the object graph as our query keywords.

In this last proposal, the effort is to make the preparation of the links completely dynamic and making the system more "flexible." Making the system more flexible is used in the sense of giving a degree of relevance to the links dynamically built. If a link is returned as the xml-link for an attribute in the previous models, it is assumed that it matches 100% and all the other links are irrelevant. The widely known inquiry technique of **"occurrence of keyword"** that simply returns the number of times the keyword has occurred in a given context consecutively, gives insight to this approach.

When a node is selected by the scenario builder to be traced back to the source of data on the JTree representation of the Suppressor object model, the selected node's String value is passed as a parameter to the *build_xml_link()* method and this method parses all the XML documents in the system, essentially walking through contents of the all document's elements. The element's content, the text between the start tag and the end tag, is parsed into tokens. If there is a matching token to the selected node's String value, the XML document's name, element's tag name and the attribute value (if it exists) are added to a collection. The frequency of how many times the selected node's String value occurs in that element's content is also important, because it tells how relevant the returned link is. It can be calculated by this formula:

$$\text{Relative Frequency} = \text{\# matchings} / \text{\# tokens}$$

In addition to the document name, tag name and the attribute value, the relative frequency is also added to the collection. The list of the xml-links plus their corresponding frequencies are

presented to the scenario builder. If there are many links returned, he may choose to view the ones which are most relevant by checking their frequency values. That helps to differentiate the values of the links returned resulting in more precise information retrieval.

3.6 CHAPTER SUMMARY

This chapter has introduced the basic steps taken to solve the problem defined in Chapter 1. As discussed in Chapter 1, the goal of this work is to develop mechanisms and applications which enable the user to trace back to the origin of data dynamically making use of Java and XML technologies. As can be understood from the discussions in this chapter, data source traceability requires a combination of various technologies, such as XML, XSLT, Xpath, Java, DOM, SAX, OODBMS, and Java Serialization. In the next chapter implementation of the methodologies discussed in this chapter is introduced.

4 IMPLEMENTATION

4.1 INTRODUCTION

This research is a study of how several technologies like XML, Java, and their derivative languages can be merged to trace the source of data for attributes of an object graph representation of a simulation scenario. Elaboration of the object model representation is provided by the work [18]. Some other technologies like OODBMS and Java object serialization are also studied to make persistent the link to source data, prepared for the purpose of traceability. The application of the methodologies introduced in the previous chapter are discussed next.

4.2 REPRESENTATIVE TEXT DOCUMENT

Due to the limitations given in Chapter 1, the text document which serves as the source of data for the object graph is not the actual one; it is a surrogate document with similar structure but fabricated content. When preparing this document it was important to represent data in a structured way so that it would allow exploration and exploitation of its content.

4.3 CONVERTING TEXT TO XML

To make text documents self-describing and advertise something about its content, the text documents need to be marked up. Owing to XML's markup capabilities (as described in Chapter 2), XML was chosen as the mark up language for this research.

In the previous chapter, there are three ways described for converting a text document into XML. By using the Java DOM API, by using eXcelon™ tools, or manually. Java DOM API was not preferred to be used in this research, because it is the most complex to use. As long as the text document is small and does not have a complex structure, it is convenient to use the DOM representation. However, for this research which has complex documents, it is not convenient. Most of the XML documents used in this work were created manually. The structure of the documents, such as the nesting rules and the multiplicities, was defined first. In accordance with

that a Document Type Definition (DTD) was developed. Then obeying the rules and the

constraints defined in the DTD, the XML document was populated with data, by placing the start

and end tags where appropriate and inserting appropriate sample text between these tags. For

most of the elements of XML documents used in this research, the attribute "id" was added,

assuming in the real document there could be a need to have the same tag name for more than one

data item. The attribute "id" serves to distinguish between elements having the same tag name. A

fragment of the "Supres.xml" document, which was created manually to represent source data in

a marked up fashion, is depicted in Figure 40.

```
<?xml version="1.0" standalone ="no" ?>
<!DOCTYPE capabilitylist [
  <!ENTITY % val "ALWAYS USE THE DEFAULT VALUE" >
  <!ENTITY longtext SYSTEM "sup1.txt" >
  <!ELEMENT cabapilitylist (capability+)>
  <!ELEMENT capability (capname, simround, platform, wpnchar,value)>
  <!ELEMENT capname (#PCDATA)>
  <!ELEMENT simround (#PCDATA)>
  <!ELEMENT platform (#PCDATA)>
  <!ELEMENT wpnchar (#PCDATA)>
  <!ELEMENT value (#PCDATA)>
  <!ATTLIST capname id CDATA #REQUIRED>
  <!ATTLIST simround id CDATA #REQUIRED>
  <!ATTLIST platform id CDATA #REQUIRED>
  <!ATTLIST wpnchar id CDATA #REQUIRED>
  <!ATTLIST value id CDATA #REQUIRED>
]>
<cabapilitylist>
  <capability>
    <capname id="some">http://afitweb.afit.af.mil/SCBY/default.asp </capname>
    <simround id="some">f22_aim9.jpg</simround>
    <platform id="some">gotoserver.ennt30yq.3000</platform>
    <wpnchar id="some"> you have to look at figure 14 in the book "WEAPONS"
    </wpnchar>
    <value id="some"> the answer is  &longtext; </value>
  </capability>
  <capability>
    <capname id="never">This value must be a String with a minimum length of  8 characters
    </capname>
    <simround id="never"> Round value is min= 6 max 9 </simround>
```

Figure 40- Manually created XML document.

Figure 40 depicts an XML document created directly in a text editor. This document has an internal DTD, which is defined at the beginning of the document as "<!DOCTYPE capabilitylist." All the rules pertaining to the structure of the document are defined in this DTD. Notice in the document that the same tag names are repeated more than once like "<capname>" or "<simround>." To differentiate between them, "id" attributes having the values like "never," and "some" are added to elements. Another important point to notice about that document is that it also tells what kind of content is going to be stored between tags. Most of the time the content is plain text, and it is this form we anticipate most scenario ground truth data will take. But the data source can also be in another format. The possible formats are a web page, an image or a table containing data. Alternatively, the data between tags can serve as an instruction, as in "<platform> gotoserver.ennt30yq:3000 </platform>." When the application developed for this research reads in that data, it realizes that the source of data is provided by the server running on **port** 3000 and on **host** ennt30yq. So the application registers with the server as a client and asks for the source data to be sent. If it is in the format of "<capname> http://afitweb.afit.af.mil/ SCBY /default.asp</capname>," a system call is made to the "IExplore.exe" passing the internet address as the parameter. Thus Internet Explorer is invoked and it brings up the web document specified by the address. It could also be in the format of "<simround > f22_aim9.jpg </simround>." This says that the data source is a jpeg-compressed file. The application has image display methods which bring up the image file to the user's view. There may arise a proposal like "instead of making a reference to the image file, let's embed it directly to the XML document." This proposal is not acceptable, because it is not possible to embed a raw graphics file or any other binary data directly into an XML file. This is because any bytes resembling markup could be misinterpreted [46].

4.4 HOW TO TRAVERSE FROM AN OBJECT GRAPH TO THE DATA SOURCE

The original object graph representing a simulation scenario is the work of McDonald.

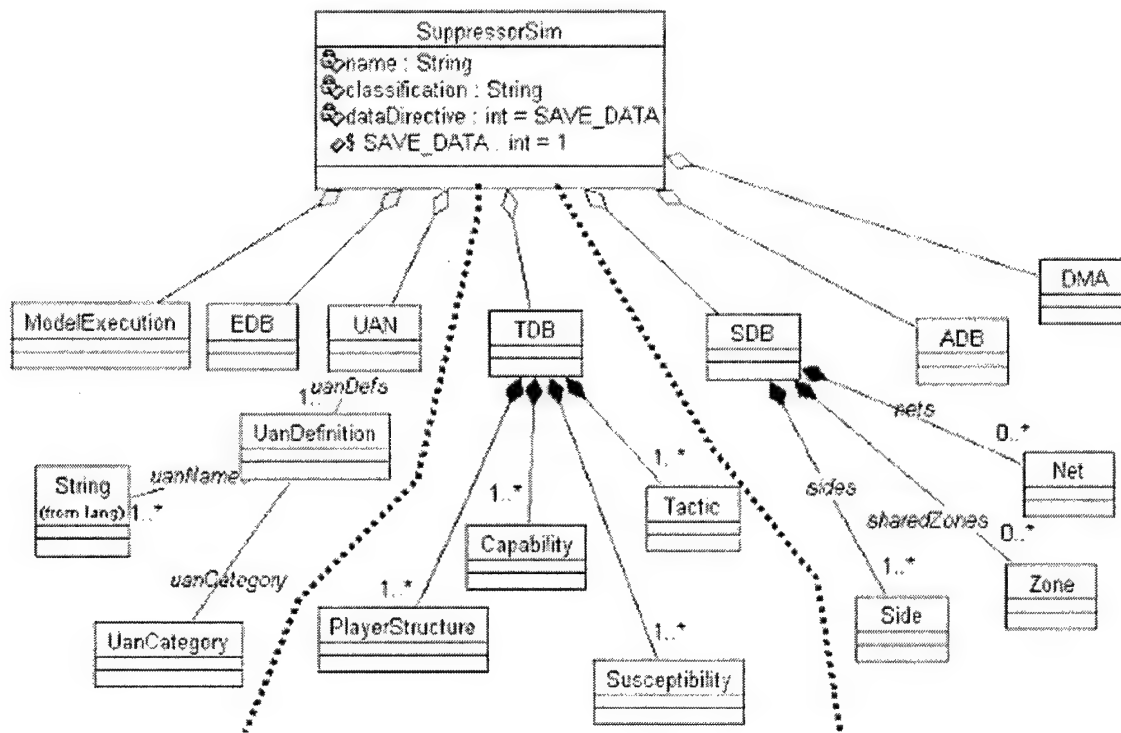Figure 41 depicts his object model which serves as the basis for this research.

Figure 41- The research area on McDonald's Object Model

As seen in the figure above a SuppressorSim object consists of UAN, TDB, SDB, ADB, etc.. The

part of the object model studied in this research area is bracketed by the dashed lines in Figure 41.

For this work TDB object and its underlying structure was chosen to be studied in a detailed way

for achieving traceability. The reason for limiting the scope of this research has two main reasons:

1- TDB is an important object in SuppressorSim object and it represents almost all the aspects of

the whole object model. Further, it has the most complex and comprehensive class hierarchy

in the SuppressorSim grammar.

2- To be able to implement all the methodologies introduced in Chapter 3. The main focus in

this research is to demonstrate as many mechanisms as possible which allow one to trace to

the origin of source data. We aim to show the concepts and discuss their advantages and disadvantages.

The figure below demonstrates the partial class hierarchy of the Susceptibility object which is one of the aggregation classes of the TDB class.
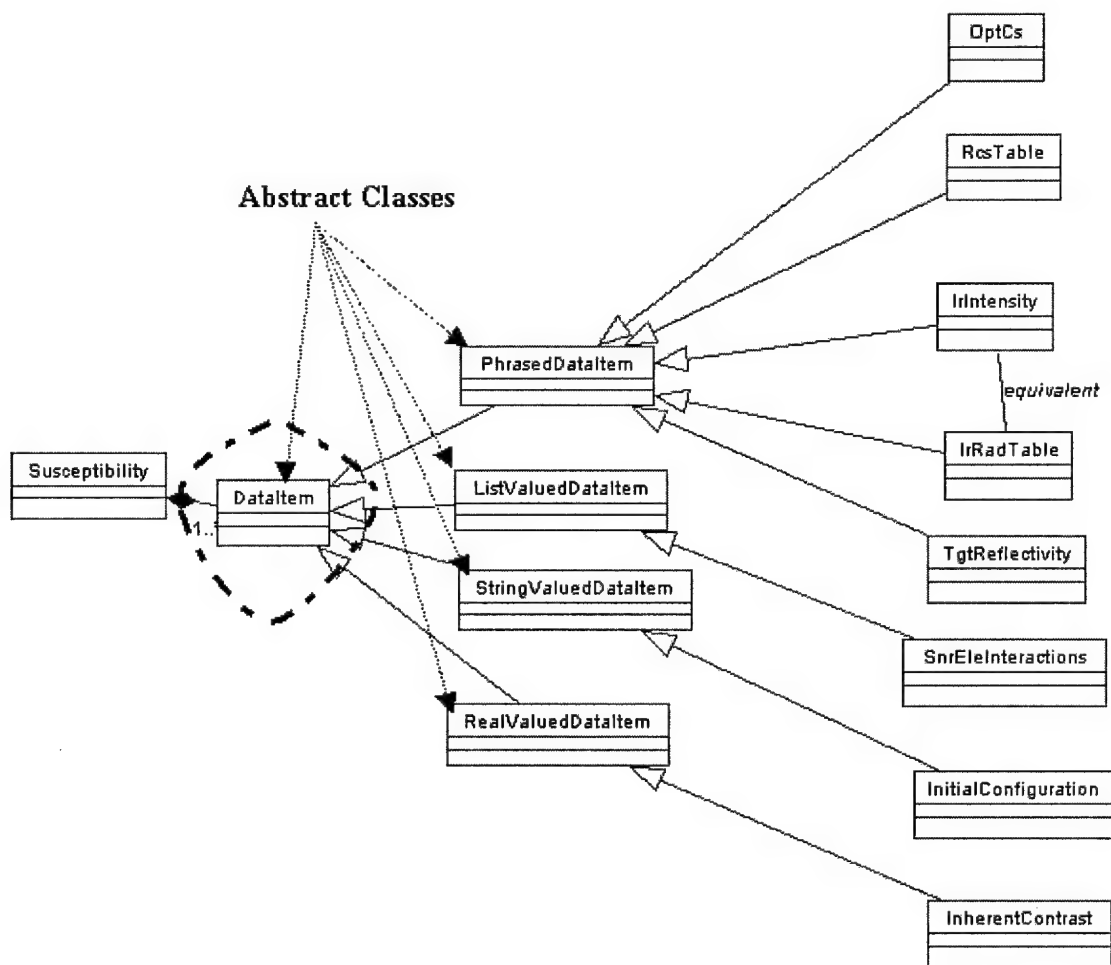
Figure 42- Partial class hierarchy of a Susceptibility object

The figure above depicts the class structure of a Susceptibility object partially; the part shown is the one most pertinent to this research area. It also gives insight into the other class hierarchies like Tactic and Capability.

The Player Structure is different from others; it gives a full description of a wide range of weapon systems and interactions taking part in the scenario. In this figure, an important point to

notice is the DataItem class (surrounded by a dashed ring), because it is an abstract class. Abstract classes provide an interface for various methods that all the classes extending this abstract class need to implement, but abstract classes themselves are not directly instantiated. Each non-abstract subclass of an abstract class has to provide the implementation for the abstract methods in the abstract class. In McDonald's DataItem abstract class, there are abstract methods like *toTree()*, *parse()* and *phrase()* whose implementations are provided by each non-abstract DataItem subclass. All the main subclasses of Susceptibility which provide the implementation of abstract methods like *toTree()* and *parse()* are the ones on the right hand-side of the Figure 42. These include: **OptCs, RcsTable, IrIntensity, IrRadTable, TgtReflectivity, SnrEleInteractions, InitialConfiguration** and **InherentConstrant.** Those classes are especially important for this research because they provide the mechanism for parsing scenario data files into a syntactic object model and ultimately for making a tree display of the object models. The following sections describe the modifications made to the methods of these classes to achieve the traceability required.

Except for the Player Structure, the class hierarchy is almost the same for Tactic and Capability. They also make use of the absract DataItem class and the main classes are the ones which extend this abstract class and which are non-abstract (concrete) classes. They provide the mechanisms for *toTree()* and *parse()* methods.

An emphasis is given to understanding the *toTree()* and *parse()* methods of McDonald, because they serve as the entry point into the application of McDonald's work to this research. These methods are the ones which are mostly modified to facilitate scenario attribute-to-source data traceability.

4.4.1 Mirror-Tree Model

The Suppressor object model is populated by parsing scenario data files as depicted in Figure 43. For example when the TDB.txt file is read, the whole file is taken as a data item block

and the contents of this single data item block are contained in the Suppressor's TDB object.
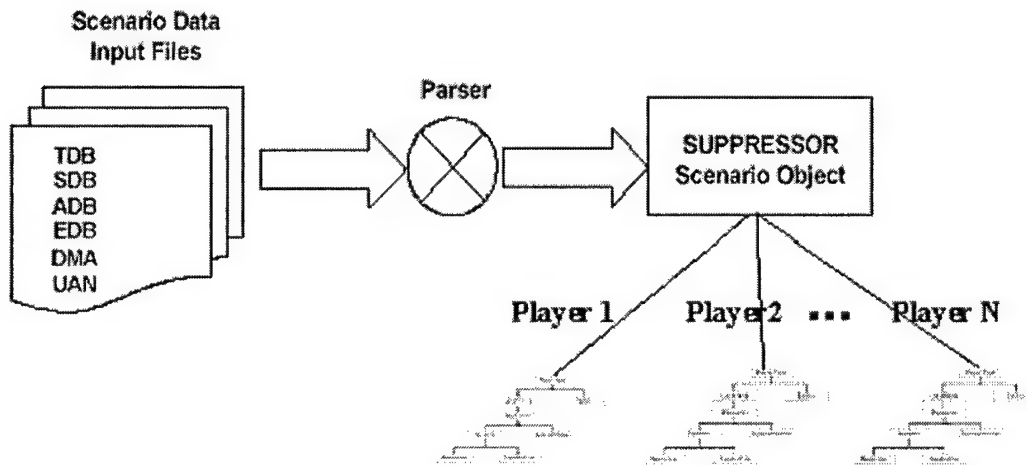
Then the parsing process starts.

Figure 43- Population of Suppressor scenario object.

Almost every subclass of TDB has a *parse()* method. Similar to the partial Susceptibility class diagram in Figure 42, a TDB class is an aggregation of the "DataItem" classes. The abstract classes like "RealValuedDataItem," "IntegerValuedDataItem," and "StringValuedDataItem" extend the super "DataItem" class. The TDB data item grammar block is broken down further into sub-data items. After that, it is determined if this sub-data item block is an instance of "StringValuedDataItem," " RealValuedDataItem" etc.. Then, the *parse()* method of the matching class is called which handles the rest of the sub-data item block. It is also possible to see that, the *parse()* method of a class tokenizes the stream and stores some of the tokens in its attributes and then passes the rest of the tokens to its subclasses' *parse()* methods. The parsing process goes on until all the stream (data item grammar block) is read into the objects of TDB.

While parsing, the correctness of the grammar of the text file is checked. However there is no verification of the tokens. Checking each token's semantic validity and ultimately intelligent scenario construction is recommended as a future work in [18].

One of the various conversion formats McDonald provides for a newly parsed and instantiated Suppressor scenario object is a Java Swing Tree representation as demonstrated in Figure 44.
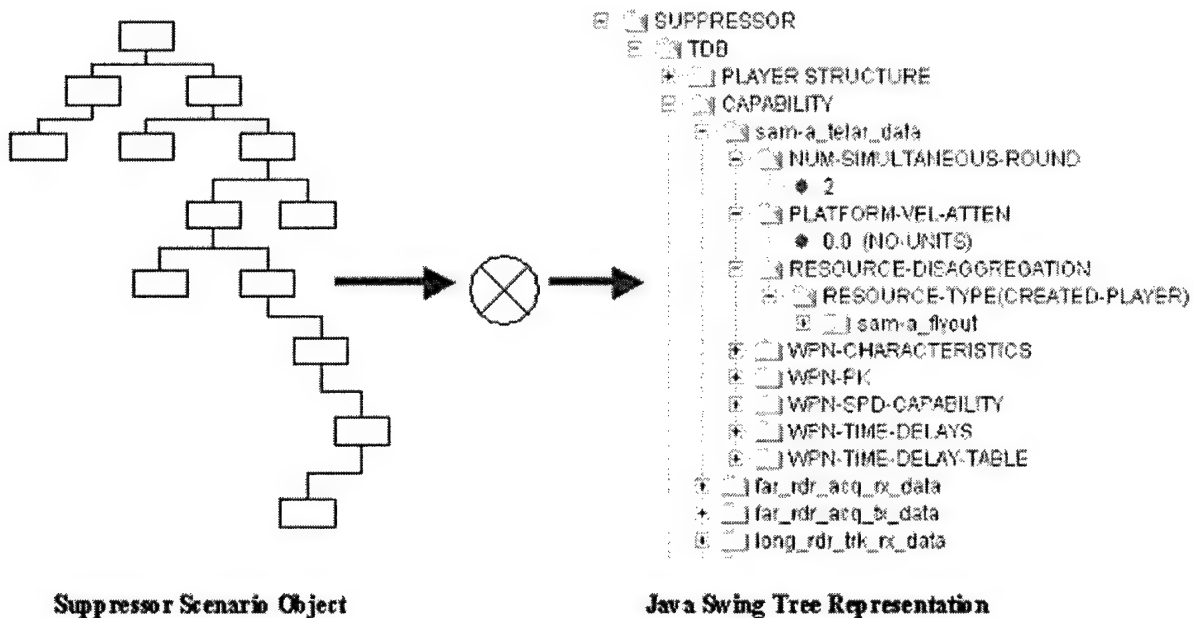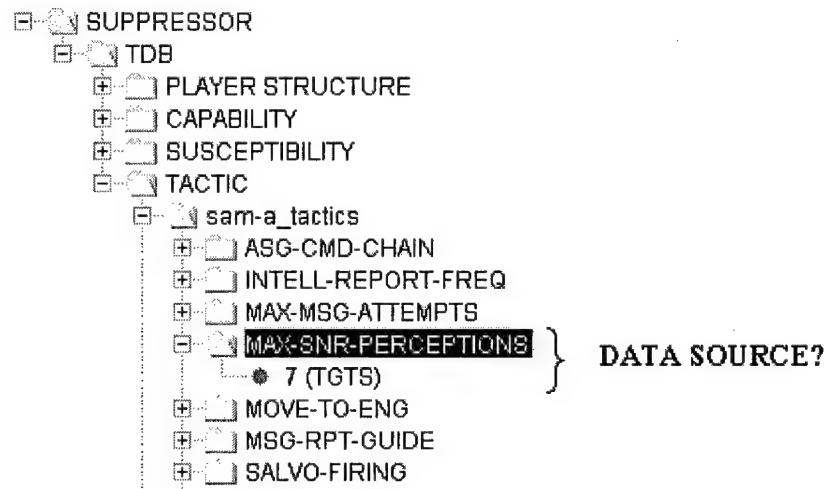

Suppressor Scenario Object   **Java Swing Tree Representation**

Figure 44- Java Swing Tree representation of the Suppressor scenario object.

The figure above shows how the scenario object is visualized as a JTree. To accomplish this, there are *toTree()* methods in most of the classes. The *toTree()* process is started at the top level object and this top level object provides the root for the JTree representation. The entire object graph is traversed by making calls to *toTree()* methods of each of the subclasses. The *toTree()* method of every subclass makes a subtree of the data item it stores and returns that subtree to the parent class that made the call to this method. In the end, the main tree is formed.

As mentioned in Chapter 3, the JTree representation provided by McDonald, despite its other utility and other aesthetic value, does not have the capability to trace back to the source of TDB data items (see Figure 45). This capability is important in the event that the scenario builder needs to make changes to the current scenario. To prepare new scenario files making use of the

old ones, he needs to see the data source to avoid semantic and grammatical input errors and to verify that he is entering the "approved" values for the new scenario.

```
☐ ⛁ SUPPRESSOR
   ☐ ⛁ TDB
      ☐ ⛁ PLAYER STRUCTURE
      ☐ ⛁ CAPABILITY
      ☐ ⛁ SUSCEPTIBILITY
      ☐ ⛁ TACTIC
         ☐ ⛁ sam-a_tactics
            ☐ ⛁ ASG-CMD-CHAIN
            ☐ ⛁ INTELL-REPORT-FREQ
            ☐ ⛁ MAX-MSG-ATTEMPTS
            ☐ ⛁ MAX-SNR-PERCEPTIONS   } DATA SOURCE?
                  ● 7 (TGTS)
            ☐ ⛁ MOVE-TO-ENG
            ☐ ⛁ MSG-RPT-GUIDE
            ☐ ⛁ SALVO-FIRING
```

Figure 45- Data source to be found.

In the figure above, we assume that the scenario builder needs to update the value for "MAX-SNR-PERCEPTIONS"; its current value is "7 (TGTS)." Suppose he enters "10 (TGTS)" for that value. But if the source of data for "MAX-SNR-PERCEPTIONS" indicates that this value can never be greater than "9 (TGTS)," this is going to cause a semantic error which may affect the results obtained when the scenario is executed, despite the syntactical correctness of the statement.

One of the mechanisms proposed for this purpose in Chapter 3 is the mirror-tree object model. In form, the mirror-tree model constructs almost the same tree structure as the original object tree. The difference is that in addition to attribute values (instances), it stores the link information to the data source of the corresponding instance on the object tree.

In this model, a new text file is prepared to store the link information to the data source (since the data source is in XML format it is also referenced as XML document). The difference between the original TDB text file and the new mirror text file is depicted in Figures 46 and 47. Figure 46 is the fragment of original scenario data text file.

```
Example # T-2:   ---  Maximum Number of Direct Perceptions  ---
MAX-SNR-PERCEPTIONS 7 (TGTS)
MOVE-TO-ENG NO
MSG-RPT-GUIDE
  DIMENSION 1 CMD-CHAIN-TYPES   intell
    REPORT-RESPONSIBILITY CMDR
END MSG-RPT-GUIDE
```

Figure 46- Fragment of scenario data file "TDB.txt".

```
Example # T-2:   ---  Maximum Number of Direct Perceptions  ---
MAX-SNR-PERCEPTIONS supres.xml-platform-someZZtezola1.xml-speed-someLL7 (TGTS)
MOVE-TO-ENG NOLLsupres.xml-platform-some
MSG-RPT-GUIDE
  DIMENSION 1 CMD-CHAIN-TYPES   intellLLsupres.xml-platform-some
    REPORT-RESPONSIBILITY CMDR
END MSG-RPT-GUIDE
```

Figure 47- Fragment of "TDB_M.txt" which has the link information.

When Figure 47 is examined, its difference from the text fragment in Figure 46 is that it has some more information which serves as the link information. For example consider "MAX-SNR-PERCEPTIONS supres.xml-platform-someZZ tezola1.xml-speed-someLL7 (TGTS)," where it is in the form of "MAX-SNR-PERCEPTIONS 7 (TGTS)" in the original text file. MAX-SNR-PERCEPTIONS is one of the subclasses of Tactic class, and it stores the data item 7 (TGTS) currently. In the modified file there are additionally two xml-links (data source information) separated by the delimiter "ZZ".

An **xml-link** is considered as the basic unit of information which the application developed for this research takes as a parameter to the traverse to the data source dynamically. An xml-link consists of three parts; a source XML document name, the tag to be found in the XML document, and the attribute value of this specific element. The delimiter "LL" separates the data item value from the xml-links.

As a consequence of altering the Suppressor grammar as we have now done, there is a need to create new *parse()* and *toTree()* methods in all of the classes in order to read in the

augmented scenario text file. These methods are named *parse_M()* and *toTree_M()* and are used to parse the new scenario grammar and consequently create the mirror tree which has the xml-link values. The "improvement" of the *parse_M()* method is that it now recognizes the grammar used in the new data file. That is, it has the ability to tokenize the augmented streams.

Similarly, the new *toTree_M()* method of a subclass of Suppressor class now tokenizes the data item value wrapped in this class. A hierarchical subtree is created making use of the tokens. The *toTree_M()* method creates exactly the same subtree as the *toTree()* method, but it adds the xml-link values to the subtree as leaf nodes. The leaf nodes of the original subtree become the parent of the new leaf nodes which represent the xml-link values.

There is also a need to check the type of the file read in. This is done in the *loadFile(File)* method of SuppressorGateway class. This method checks to see if the file is the original text file or the augmented text file. If it is the original one, then when parsing the file and visualizing the object model *parse()* and *toTree()* methods are called, respectively. If it is the augmented one, then when parsing the file and visualizing the object model *parse_M()* and *toTree_M()* methods are called, respectively. The result of this approach is depicted in Figure 48.

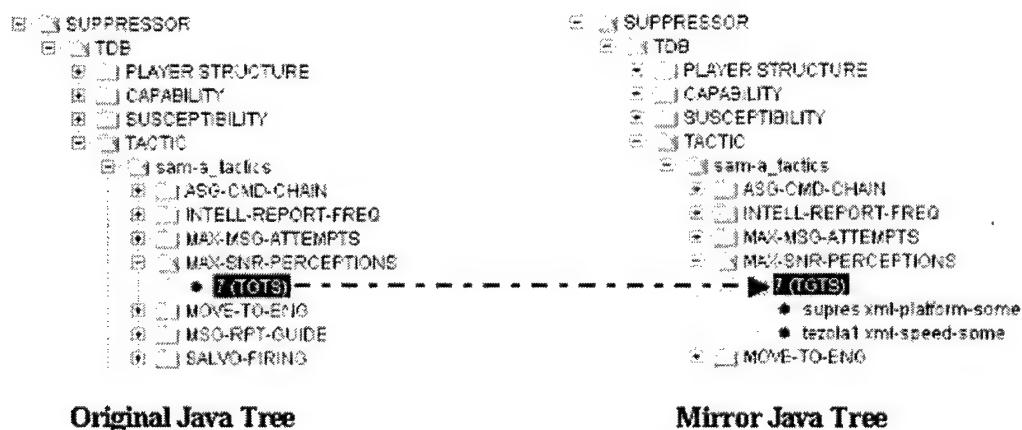Original Java Tree   Mirror Java Tree

Figure 48- Original Java Tree and Mirror Java Tree.

Figure 48 gives a visual expression of the implementation of the mirror-tree model discussed up to this point. The data sources of data item value "7 (TGTS)" of the object "MAX-

SNR-PERCEPTIONS" are "supres.xml-platform-some" and "tezola1.xml-speed-some." There is

a need to develop some mechanisms which traverse from the original Java tree to the mirror Java

tree and find the xml-link values stored there. Further, a mechanism is required to pass the xml-

link values to some other mechanisms that return the related fragment of the marked up data

source XML document. The implementation of these concepts is discussed next.

4.4.1.1 Finding XML-Links in the Mirror Java Tree

On the original JTree, when the scenario-builder selects a node to trace back to the origin

of source data, first the *getPath()* method provided by the **DefaultMutableTreeNode** class in the

Java Swing Tree package is used. This method returns the path to the root as an array of tree

nodes. The path value in that node array is the main parameter used to traverse the mirror tree and

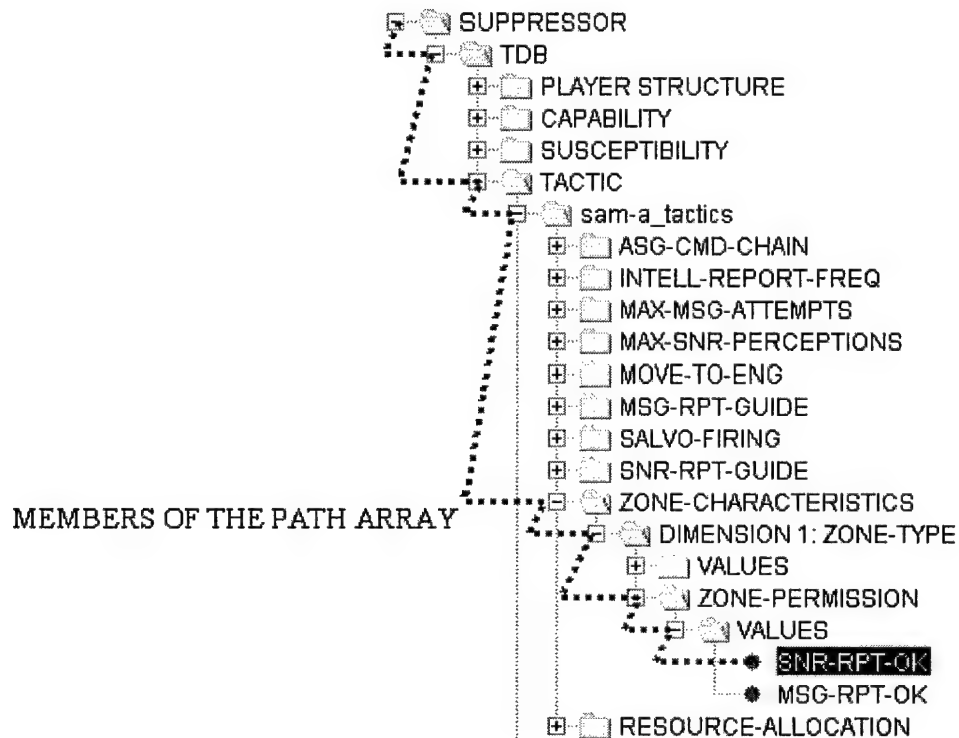find the corresponding xml-links. Examine the figure below.

Figure 49- Elements of the path array of the selected node on the original JTree.

In Figure 49, the scenario builder selects the node "SNR-RPT-OK" to trace back to the origin of source data. After this selection, the *FindSource()* method is called. This method first makes an array of the nodes which form the path from the selected node to the root of the tree by using the *getPath()* method described in the previous paragraph. The elements of the path array built by this method are shown by the dashed lines. The elements of the path array (and this is the actual order of elements in the path) are: "SUPPRESSOR," " TDB," "TACTIC," "sam-a_tactics," "ZONE-CHARACTERISTICS," "DIMENSION 1: ZONE-TYPE," "ZONE-PERMISSION," "VALUES" and "SNR-RPT-OK."

The first thing done after this is to call the *getJtreeModel_M()* method, which returns a handle to the mirror JTree which has the xml-links. There is no native method in Java which takes the path array as an input and traverses to the node following the path array on a given JTree. So a traversal mechanism which traverses to a specific node on a JTree by making use of the path array elements was developed by this author. How this mechanism works is presented next.

The first element in the path array is always the root of the JTree, so the first element in the path array and the root of the mirror JTree are compared. If they are matching, then the process continues. Then the native *children()* method of the Java Swing Tree, which returns the enumeration of a node's children, is called to find the children of the root node. The matching child node to the second element of the path array is found. Then the children of the child node are found and they are compared to the third element in the path array. This process continues recursively until all the elements in the path array are evaluated in this way. The last element in the path array is always the node selected to be traced back to the origin of source data. On a mirror JTree, the xml-links are always added to the leaf nodes of the original JTree as the children of these leaf nodes. Consequently, all the children of the last node found are the xml-links. Figure 50 depicts this action.
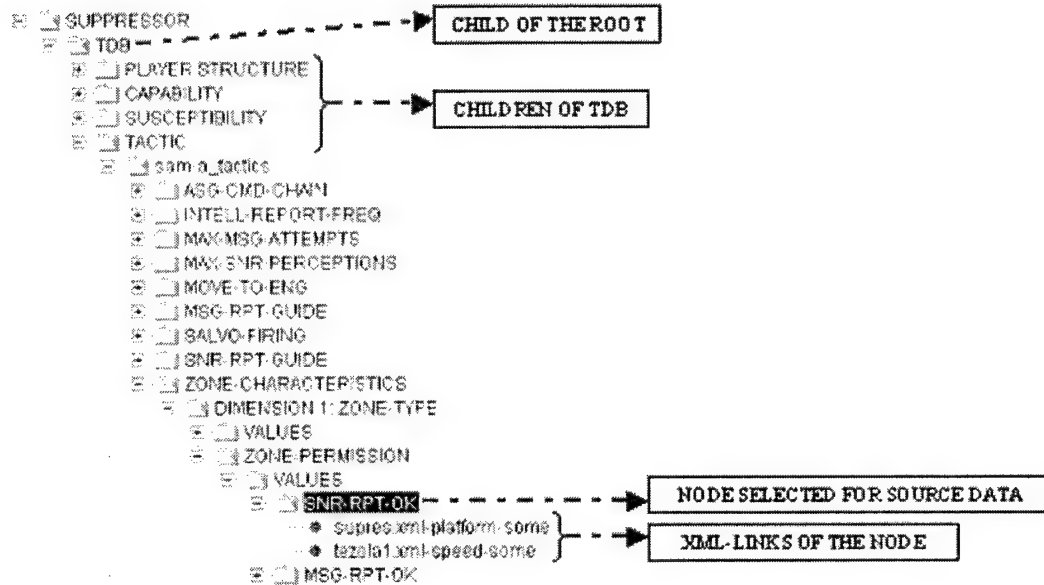
Figure 50- Traversal on the mirror JTree.

In Figure 50, the TDB node is the child of the SUPPRESSOR node, and the four nodes PLAYER STRUCTURE, CAPABILITY, SUSCEPTIBILITY and TACTIC are the children of TDB as shown. When following the path array, to the children of TDB, the traversal mechanism finds the node matching the TACTIC element in the path array, which is one of the nodes on the path to SNR-RPT-OK. This process continues recursively until the selected node is found. The xml-links are the children of the SNR-RPT-OK node. After the xml-link values have been found, another mechanism takes over the job to fetch the related fragment of the XML document to the user. The details of this mechanism are discussed next.

4.4.1.2 Mechanism to Fetch XML Document Fragments

An xml-link has three important parts; XML document name, tag name, and the attribute value of the specific tag. The first thing is to locate the XML document. Once the XML document is found, it is parsed by JAXP 1.0. Meanwhile a validity-check (whether the XML document conforms to the DTD specified or not) is also performed on the document by the XML parser.

74

The parser also builds a DOM of the XML document. By using the "*getElementsByTagName(String tag_name)*" native method in "org.w3c.dom" package, the node list of the elements in the XML document having the specified tag name is returned. To differentiate the tags having the same tag names, elements have "id" attributes. So in order to get a more precise result, another method, "*getAttribute("id"),*" in this package is used, and this method returns the value of the "id" attribute of the element. The value returned is compared to the attribute value in the xml-link and if they are matching, the element which has the source data or a pointer to the source data has been found. Finally, performing the native method "*getNodeValue()*" returns the contents of the element found. Figure 51 illustrates this process.
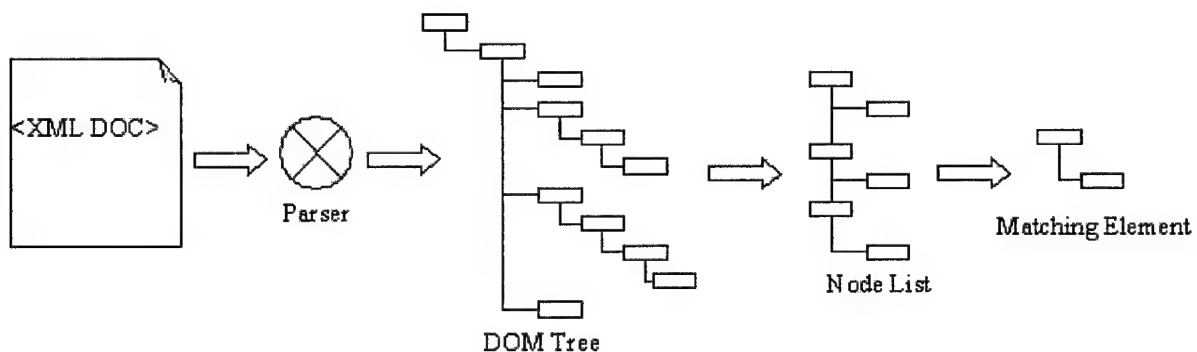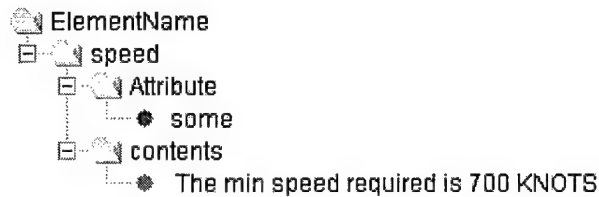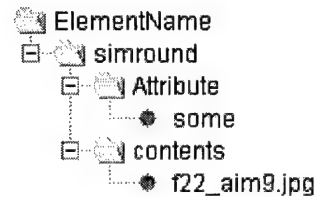

Figure 51- Extraction of source data from the XML document.

In Figure 51, when the XML document is parsed, a DOM tree is built first. By using methods in "org.w3c.dom" package this DOM tree is manipulated. As a result of using "*getElementsByTagName(String tag_name),*" the node list which has this tag name is formed. Lastly, by calling the "*getAttribute("id")*" method, the matching element is found.

The matching element and its contents are returned to the user in a display window as a subtree which may have various formats as in Figure 52.
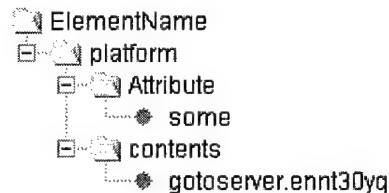
```
ElementName                                ElementName
  speed                                      simround
    Attribute                                  Attribute
      ● some                                     ● some
    contents                                   contents
      ● The min speed required is 700 KNOTS      ● f22_aim9.jpg
```

(1) Data Source in Plain Text Format   *(2) Pointer to the Data Source*
as the name of an Image File

```
ElementName                                ElementName
  capname                                    platform
    Attribute                                  Attribute
      ● some                                     ● some
    contents                                   contents
      ● http://afitweb.afit.af.mil/SCBY/default.asp   ● gotoserver.ennt30yq
```

(3)Pointer to the Data Source   *(4) Pointer to the Server which*
as a Web Page   *Computes the Data Source*

Figure 52- Various formats the data source can have.

4.4.2 Link Builder Model

Section 4.4.1 describes much of the details of the mechanisms used to support the scenario builder. This section does the same for a person responsible for creating and making persistent the xml-links (detailed information on persistence mechanisms for the xml-links is provided in Appendix B) associated with Suppressor scenario attribute values. The scenario builder acts essentially as the client, asking the server (link builder) to create the corresponding xml-links for each attribute value of the Suppressor scenario object. The link-builder acts as the server which has the capability to create the corresponding xml-links.

In the implementation of this model, methods in the "java.net" and "java.io" packages are used. First, a figure depicting the sequence of events in this mechanism is shown, then an elaboration of how this mechanism works is introduced next.
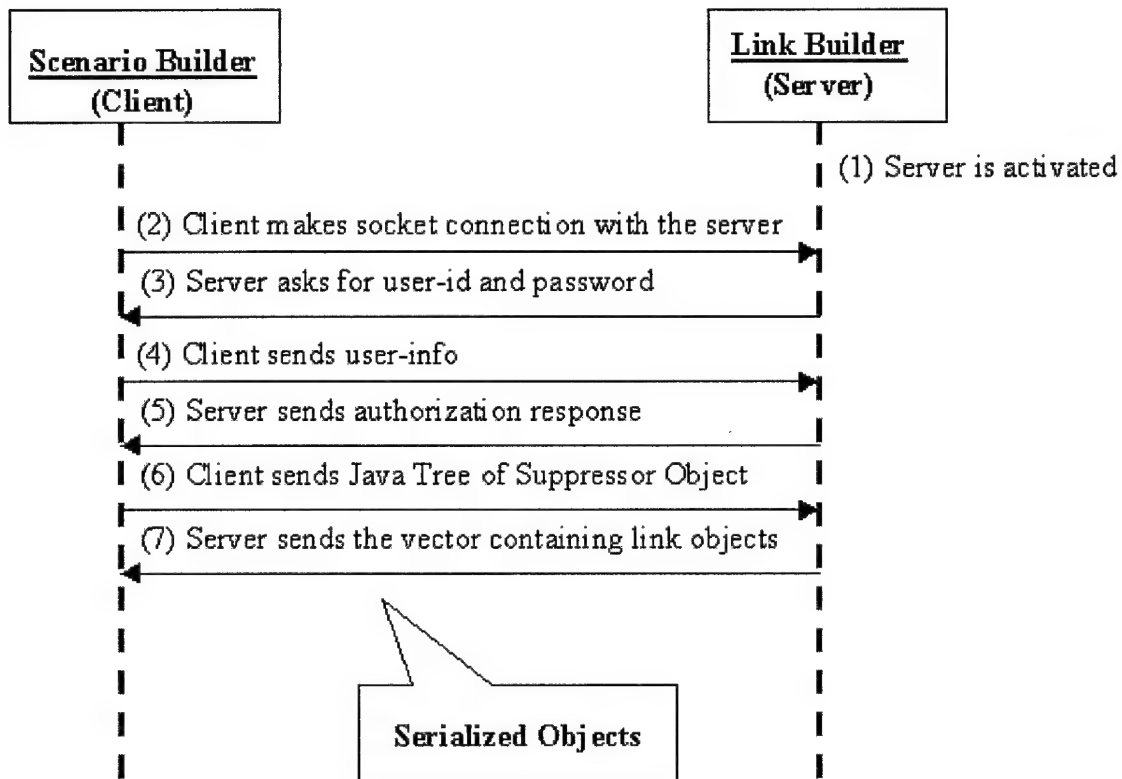
Figure 53- Sequence of events in the link builder model.

As in Figure 53, the server and the client are on different machines. When the server is activated on another machine, it waits for clients on the default port number 4000. The scenario builder (the client) makes a connection with the link builder (the server), either through agent conversations or direct socket connections, depending on the sophistication of the implementation. (Both the implementation and the demonstration of the **integration** of the link builder model with the agent based framework established by [18] is presented in Appendix A.)

After the connection is established, the server checks to see if the scenario builder is authorized to access xml-links information, and asks for the user-id and password. The scenario builder prepares its authorization message. The prepared message has three String objects; **message id** (this is fixed as "user-info"), **user-id,** and **password**. In this work, the three String components of the authorization message are put in a vector (this can also be achieved by creating a message object and mapping these three String objects to the attributes of the message object)

and this vector object is written to *ObjectOutputStream* by using *writeObject(Object obj)* and sent to the server over the connection. The server receives the message by invoking the *readObject()* method on the *ObjectInputStream*. The received object is cast as a vector object and the first element of the vector (the message-id) is checked to decide on the action to be taken. Since it is "user-info," the server checks if the second (user-id) and third (password) elements in the received vector are present in its list of authorized users. If so, the response is sent back to the client. At this point, either the connection continues or it is finalized.

If the scenario builder (client) has the authorization, it asks for the xml-links to be created by the link-builder (server). The JTree representation of the Suppressor object and a String object which has the value "tree-info" are put in a vector and this vector object is written to the *ObjectOutputStream* and sent to the link-builder. The link builder receives this message by reading the *ObjectInputStream*. The first thing done is always to check message-id generally fixed as "tree-info." The link builder casts the second element in the received vector as JTree and makes it visible on a JFrame. Then a person in charge of preparing links on that machine goes through all the JTree and adds xml-link values for all the leaf nodes (which are assumed to represent attribute values in this research) on the JTree. When he selects a leaf node, a dialog pane appears and he is asked to enter the name of the source XML document, the tag name in this document, and the attribute value of the element. When he moves on to another node, the path of the previous node to the root of the JTree, the values submitted via the dialog pane are mapped to a link object dynamically. Every time a link object is created it is added to a vector. When the person creating links completes preparation of the links, the vector object, which has all the link objects, is written to the *ObjectOutputStream* and sent to the scenario builder.

When the vector of link objects is received by the scenario builder, it is mapped to a variable. The link objects in the vector are manipulated by using get/set methods. At this point, the user can select a node to trace back to the origin of source data on the original JTree representation. To find a data source, the path of the selected node to the root of the JTree is

passed to the mechanism as the main parameter. This mechanism initially goes through the link objects in the vector and compares the passed path parameter to the **"path attribute values"** of the link objects. If a matching is found, the link object's XML document name, tag name, and attribute value are passed to the XML parser object. The rest of the mechanism works the same as presented in the section 4.4.1.2.

The link builder model implemented in this section and the mirror tree model implemented in section 4.4.1 have some drawbacks which were discussed in section 3.4.3. A third model, **meta-class instance model**, was developed to overcome the drawbacks of the first two approaches. The implementation of this model is introduced in the next section.

4.4.3 Meta-Class Instance Model

In the implementation of the meta-class instance model introduced in section 3.4.4, a new object model which has the same class hierarchy as the original Suppressor object model is created. This idea is depicted in Figure 54.
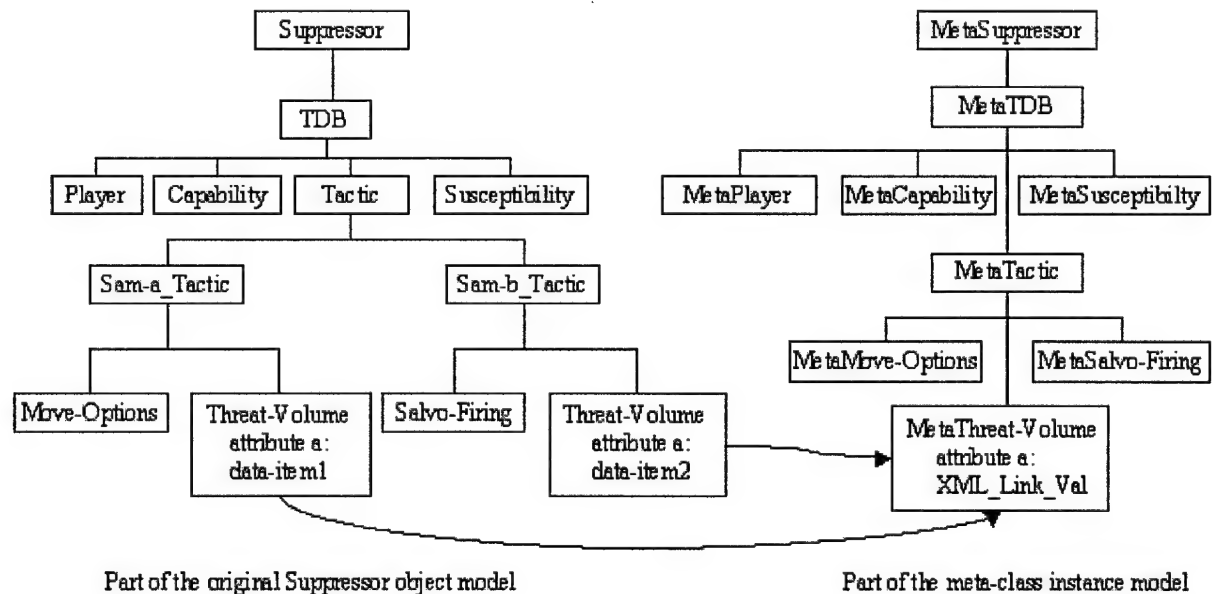

Part of the original Suppressor object model   Part of the meta-class instance model

Figure 54- The mapping between the two class hierarchies.

79

Several differences between the two class hierarchies are apparent. The original object model has instances of the Tactic class like "Sam-a_Tactic" or "Sam-b_Tactic." An instance of a Tactic class may have up to thirty subclasses. In the figure above, for demonstration purposes, the two instances of Tactic are set to have only two subclasses.

In McDonald's object model, the stream tokens read from the scenario data files are stored in the subclasses of the instances of the four main classes; Player Structure, Capability, Tactic and Susceptibility, as data items. The data item value may be for example "67.25 SEC" or "THREAT-AVOID." The pointer to the data source of each of these data items is kept in a separate meta-class instance model. The MetaTactic class does not have instances, instead it has all the thirty possible subclasses. Xml-link values are stored in these subclasses as attribute values. The initial values of the xml-links are hard-coded to known default sources, but they are editable, as may be necessary when ground truth data sources change or are updated.

There are two Threat-Volume instances of Tactic in the original object model. The data-item1 may have the value "threat-avoid-yes" and the data-item2 may have the value "threat-avoid-no." If the scenario builder wants to traverse to the source of data for these specific data item values, as shown in Figure 54 by arrows, the mapping object in the meta-class instance model is the same Threat-Volume object for both of the data items. The mechanism for finding the mapping object in the meta-class instance model is discussed next.

4.4.3.1 The Mechanism to Traverse to Meta-Class Instance Model

Figure 54 may give the insight that the traversal is from the Suppressor object model to meta-class instance model. Actually, what is implemented in this work is to traverse from the JTree representation of the Suppressor object model to the meta-class instance model. The invocation of the mechanism is almost the same as the previous models for which the implementation was given in sections 4.4.1. On the JTree representation, the scenario builder selects the node representing an attribute for which to find the source of data. The path from the

selected node to the root of the JTree is found by the *getPath()* method which returns the path array. The path array has the pointers that guide the traversal through the meta-class instance model.

What is done is implementing a **Deterministic Finite Automata (DFA)**, a simple computing machine that handles the traversal in the object model. In [47], a DFA is defined as a finite state machine having:

1- A finite set of states.
2- A finite alphabet or input symbols.
3- The initial state.
4- The set of accepting states.
5- The transition function.

In the case of the traversal of the meta-class instance model:

1- The finite set of states is the all the objects in the meta-class instance model. Every object represents a state and upon arriving at an object (state) a decision is made (or may be the final point is reached)

2- The finite alphabet is all the nodes in the JTree representation of the Suppressor object model.

3- The initial state is always the Meta-Suppressor object.

4- The discussion below gives the explanation for both 4 and 5.

When a node is selected on the JTree to traverse to its source of data, a path array is formed as discussed in section 4.4.1.1. The set of accepting states is the path array. It serves as the guide through the traversal of the meta-class instance model. The first element in the path array is always the root of the JTree, so it always leads to the initial state (Meta-suppressor object). After this point, the decision as to which object to traverse next starts. The pointer is on the second element of the array. This second element in the path array helps make the transition from one object to the other matching object in the hierarchy correctly. This is implemented by comparing the "pointed to" element of the array to the strings hard coded as the values required for the transition from one object (state) to an other (state).

When the transition from one object to another object is done successfully, the pointer shifts to the next element in the path array. (If no matching (next) object is found, then the function exits). This comparison (of the current element of the path array to the required transition values for the meta-class instance model's objects) and the transition from one object to another, continues until the object which has the xml-links is found. This concept is illustrated in Figure 55.
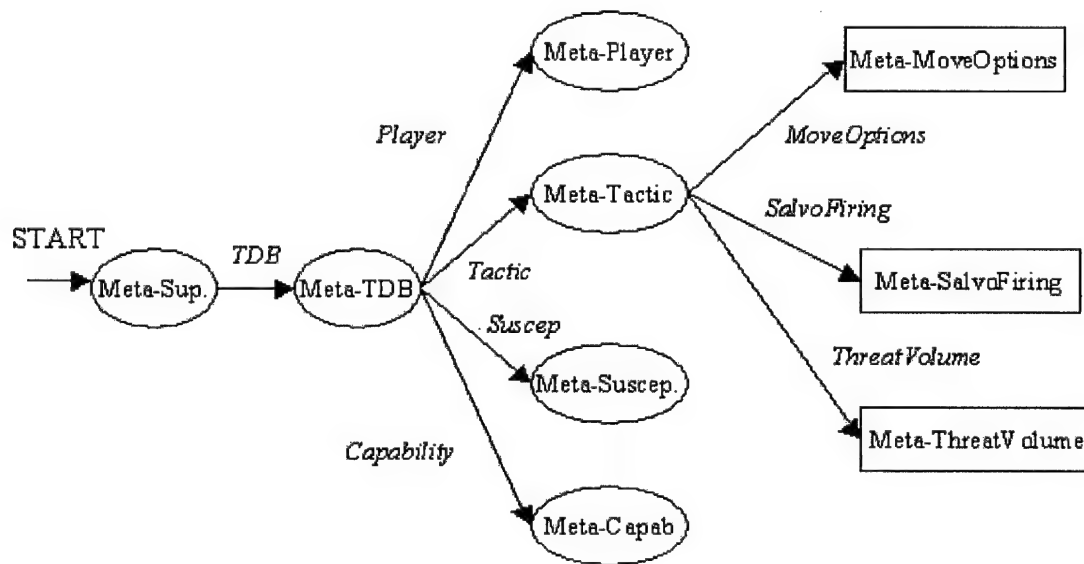

Figure 55- Part of the DFA established.

In the figure above, "Meta-Sup" is the initial state. The DFA always starts at that point. The pointer in the path array then shifts to the next element in the array. If it is equal to "TDB", then the Meta-TDB object is the next object and the pointer in the path array shifts to the next element. These are implemented by a *go-next-Object()* method. To move forward, the pointed element of the array has to be equal to one of the four cases: "Player," "Tactic," "Suscep," or "Capability." This process continues until the final states (rectangular shaped states) are accessed. The rectangular shaped objects indicate the components which contain the xml-link values. The xml-link values are then passed to the XML parser object, and the rest of the mechanism works the same as presented in the section 4.4.1.2.

In all the models implemented until now, DOM API is used to manipulate the data stored in XML documents. An alternative approach was implemented, and is discussed next.

4.4.3.2 XSLT Model

As introduced in Chapters 2 and 3, XSL has two primary functions; providing a set of tag-based transforms for the contents of an XML document and giving style to XML documents. These abilities give insight to implementing XSLT as an alternative technique to DOM for manipulating XML documents. Our meta-class instance model is extended for that purpose. To the objects of the meta-class instance model, one extra "xslt-info" attribute was added as shown in Figure 56.

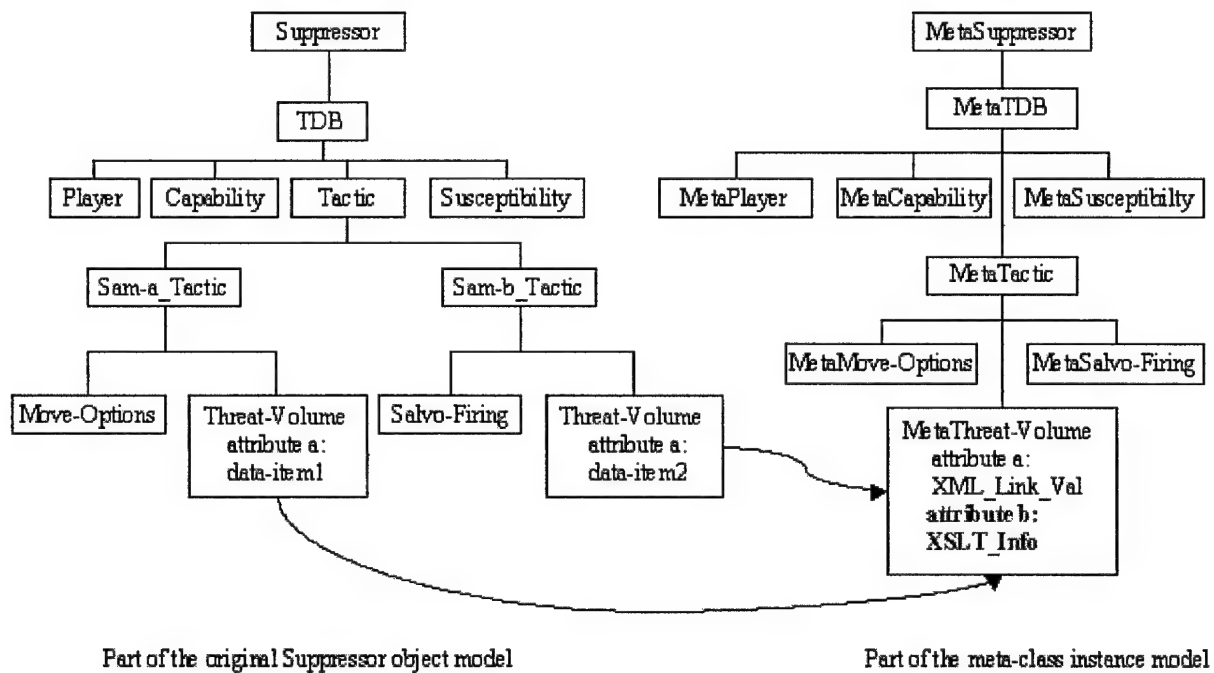Part of the original Suppressor object model   Part of the meta-class instance model

Figure 56- The extended meta object model to include XSLT-info.

The XSLT-info stored in meta objects helps create XSL dynamically. The XSLT-info consists of the XML document's name, the Xpath to the specific element in the document, and the attribute value of the element. Every time the scenario builder selects a node in the original JTree representation, the XSL file is overwritten with the XSL-info in the meta object. Once the

83

name of the XML document is known and the XSL is written out, then there is a need to activate the browser within Java code. This is needed to automatically apply the style and the transformation defined by the XSL on the XML document, and view it. This is achieved by using the methods in the Runtime class of "java.net package." The final form of the source data is presented to the user in the browser window. Figure 57 depicts how this is implemented.
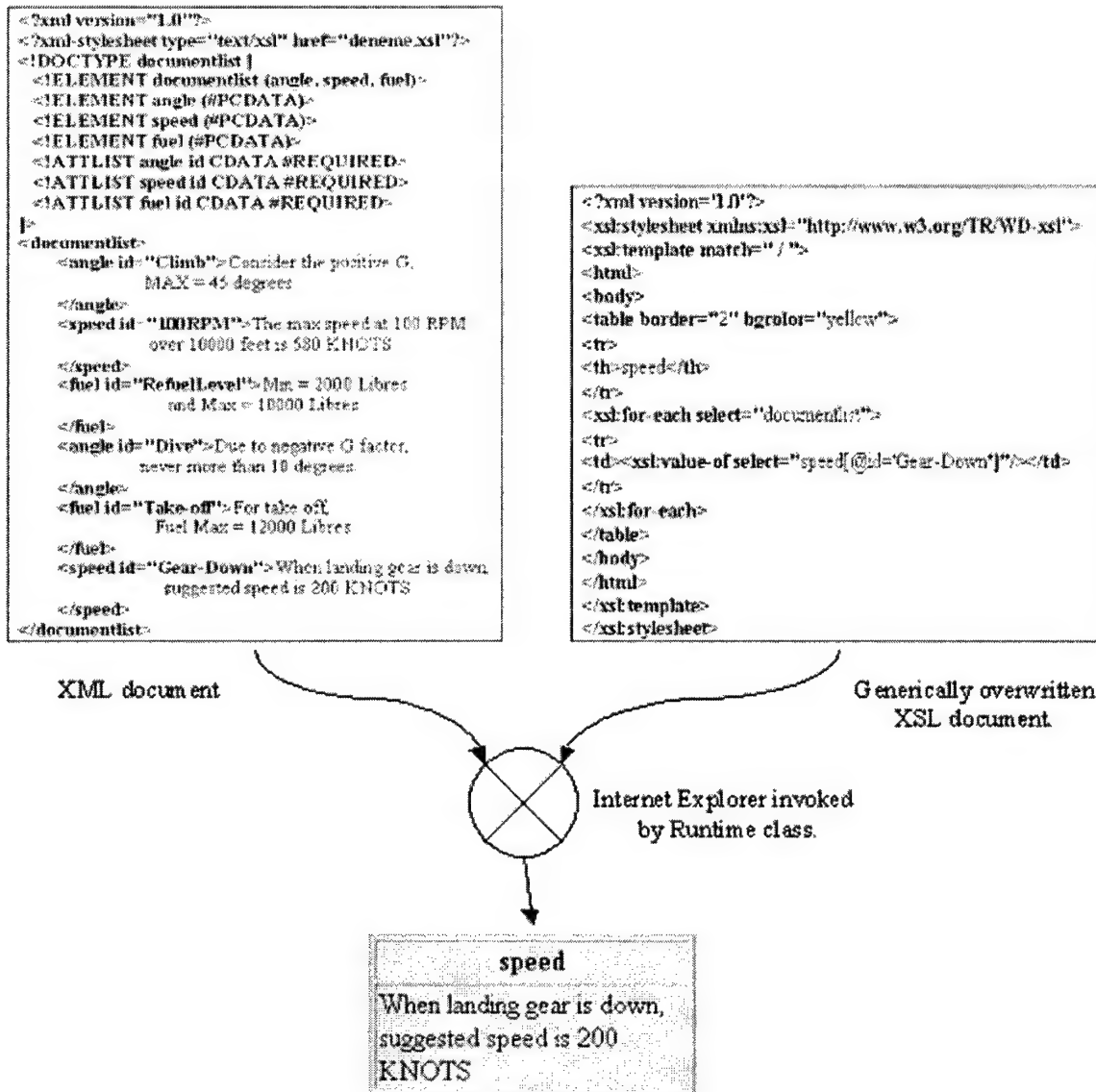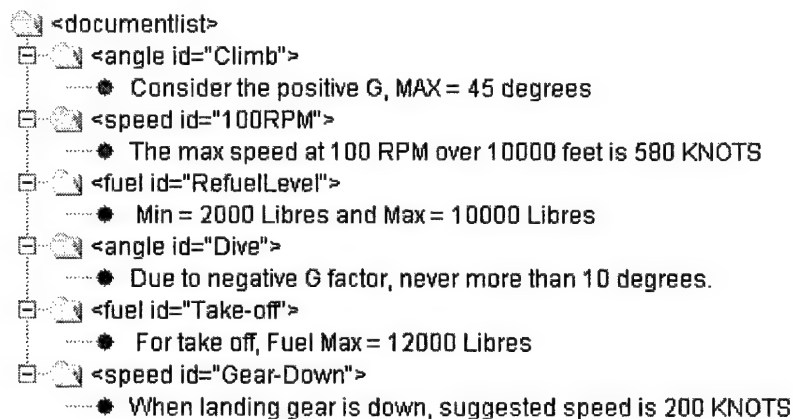

Figure 57- XSLT implementation.

In Figure 57, the XSL is generically written out by making use of the XSLT-info in the meta-object. To create the specific XSL in this figure, the XSLT-info is "flightInfo.xml*speed &documentlist&Gear-Down", where "flightInfo.xml" is the document to be transformed, "speed" is the tag name, and "documentlist" defines the Xpath to reach the "speed" element's contents. Since there is more than one "speed" element in the actual source document, "Gear-down" is the distinguishing value of the "id" attribute of speed. These are the main variables needed to create the XSL in this figure. Finally, making a system call to the browser application brings up the data source to the scenario builder. Thus, the user may view the attribute's source data in its nature document in a browser.

4.4.3.3 Editing the Meta-Class Instance Model's Link Values

The meta objects have default xml-link and xslt-info values initially fixed in the implementation code. When the user wants to traverse to the data source, he first views the documents pointed to by the default values. Most of the default xml-links have only the document name, like "flightInfo.xml." For example, for the XML document in Figure 57, the initial JTree representation of the DOM of the XML document is presented to the user as in Figure 58.


```
<documentlist>
  <angle id="Climb">
      Consider the positive G, MAX = 45 degrees
  <speed id="100RPM">
      The max speed at 100 RPM over 10000 feet is 580 KNOTS
  <fuel id="RefuelLevel">
      Min = 2000 Libres and Max = 10000 Libres
  <angle id="Dive">
      Due to negative G factor, never more than 10 degrees.
  <fuel id="Take-off">
      For take off, Fuel Max = 12000 Libres
  <speed id="Gear-Down">
      When landing gear is down, suggested speed is 200 KNOTS
```

Figure 58- JTree representation of the DOM of an XML document.

When the scenario builder views the JTree in Figure 58, he may change the xml-link value and make it more specific. Reading the information in the JTree, if he decides that one of the elements is the exact data source, in addition to the XML document name, he can add the tag name of the element and its unique identifier attribute value. It is possible to change the xslt-info values almost in the same way.

When the xml-link and xslt-info values of the meta object model are updated, it is important to save the state of this object model for future use. This is done in two ways. State saving can be achieved by using the Java serialization mechanism. The state of the meta object model is written out to a file and when needed later on, this specific file is read in and the meta object model is restored. Alternatively, the state of the object model may be persisted by using an OODBMS, such as ObjectStore. The meta object model is inserted into the database by its root object and is thus made persistent. It is possible to manipulate the meta object model in ObjectStore as if it were an in-memory collection of objects. The entry point to the persisted object model is the root of the object model.

To extract link values from the meta object model in the OODBMS, a session is established and the specific database to which the desired meta object model has been saved is opened. The xml-link or xslt-info are extracted from the meta objects by using the same mechanism described in section 4.4.3.1

As discussed in Chapter 3, using ObjectStore is more efficient for several reasons: it is faster than serialization, the performance of manipulating objects in ObjectStore is the same as manipulating in-memory objects, and it also ensures integrity of data and reliable data management as a result of being a DBMS.

4.4.5 Information Retrieval Aspect

This concept introduced in Chapter 3 is not actually implemented in this work, it was introduced to give some insight into future work.

4.5 CHAPTER SUMMARY

The mechanisms implemented to trace back to the source of data are **mirror-tree model, link builder model** and **the meta-class instance model**. When all three approaches were implemented and compared, the meta-class instance model was found to be the most efficient one.

The mirror tree model and the link builder are specific to one scenario file. That is to say, for every scenario file, a corresponding mirror tree model or a link builder model has to be implemented. This is not true for meta-class instance model. This is because the meta-class instance model is not as strictly dependent as the mirror tree model and the link builder model on the path array (which consists of the nodes from the root of a JTree to a selected node). Thus, when the meta-class model is built once and stored in ObjectStore, it works well with any scenario file.

The drawbacks of the mirror tree and link builder model and the advantages of the meta class instance model discussed in section 3.4.3 are made clear as a result of the implementation of each of the approaches. The demonstration of the implemented models is made in the next chapter, to provide the reader with the author's implementation results.

5 DEMONSTRATION

5.1 INTRODUCTION

This chapter consists of three main sections; descriptions of the development environment, the runtime environment and the application environment. They are introduced respectively in the following sections.

5.2 DEVELOPMENT ENVIRONMENT

All the applications in this work are developed using a Windows NT environment running on a Pentium-class workstation. To create the code in the applications, IBM Visual Age for Java, Professional Edition 3.0 (Early Adapter's Environment) is used. It is important that it should be Early Adapter's Environment, otherwise the private methods and the private fields defined in the applications are not visible.

5.3 RUNTIME ENVIRONMENT

The runtime environment for the applications of this work is any platform that supports the standard Java Virtual Machine (Java Development Kit (JDK) 1.2.1 or higher). There are three software packages for the applications; details, meta and test. The details package contains the base classes for XML document parsing. The meta package contains the classes created to construct the object model for the meta-class instance model. There is a need to have the meta object model's classes in a separate package, because this object model is stored in Object Store and to store an object model in ObjectStore, all the classes in the model have to be persistence capable. These meta classes could also be made persistent capable if they were in the test package for example. But this time there would be a need to explicitly name all the classes in the batch file (used to compile the Java files) to make the compiled classes persistent capable. This is error-prone and it could cause some problems as the number of classes to be made persistence capable

increases. The test package has the base classes that allow the parsing of a set of Suppressor scenario data files.

In order for the applications to work correctly, there are third-party libraries needed. Java XML Parser (JAXP) 1.0 class libraries (by Sun Microsystems Inc.) are needed to be in the class path of the Java Virtual Machine to have the XML documents parsed and manipulated by DOM, SAX or XSL. Another library need to be in the class path of the Java Virtual Machine is ObjectStore API classes. This library is needed to allow the OODBMS server features to work correctly.

The application is developed in Visual Age but in order to run all the applications, the packages need to be exported outside the Visual Age. The exported Java files in the packages are compiled and run using batch files.

5.4 THE APPLICATIONS DEVELOPED

To start all the applications in this work, the first thing to be done is to populate an instance of the Suppressor object model by selecting the **load** menu item from the **file** menu as shown in Figure 59.
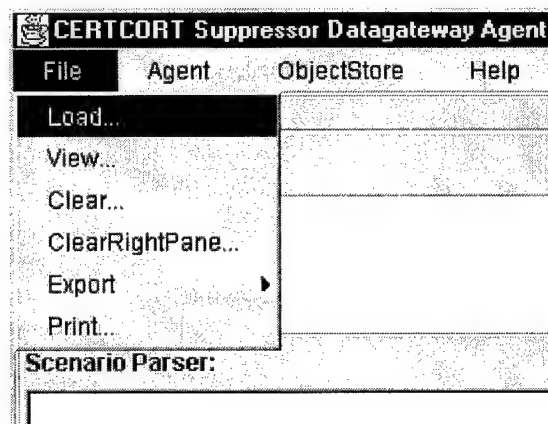
Figure 59- Selection of the Load menu item.

The selection of the **load** menu item in Figure 59 brings up the file-chooser in Figure 60.
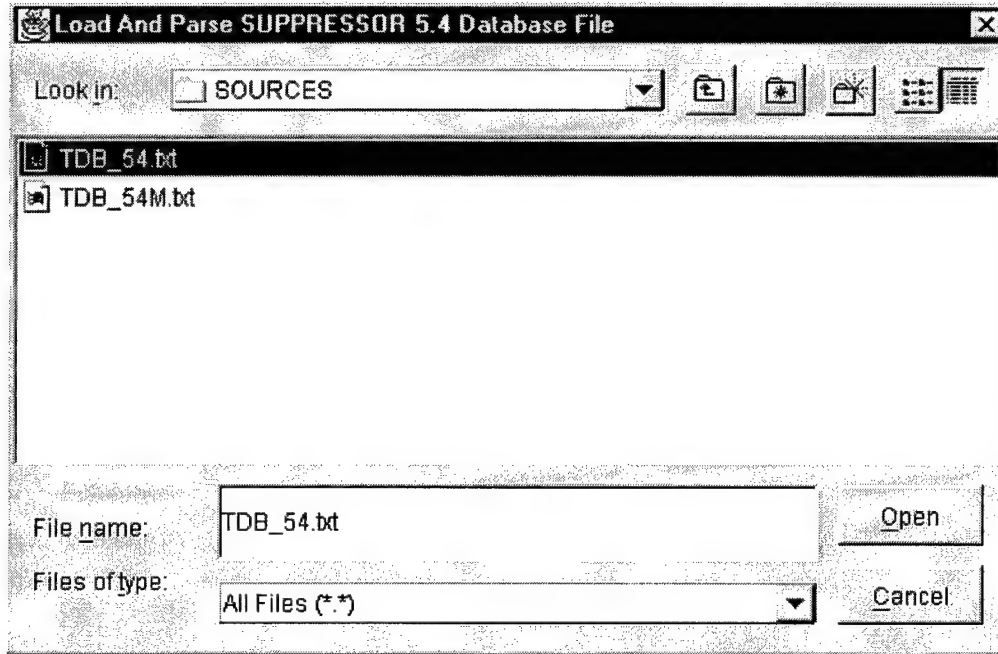
Figure 60- Selection of the scenario data file.

As in Figure 60, the TDB.txt file is chosen to populate the Suppressor object. When the **open** button is selected, the parsing of the data file and population of the object model starts. After the object model is populated, the data items stored in the Suppressor object model are visually represented in a hierarchical fashion as a JTree on the left pane of the frame as depicted in Figure 61.

In Figure 61, the "MAX-MSG-ATTEMPTS" class has the current value "4". At this point the ground truth data source is not known (or at least not indicated in the instance) for this particular value. The models demonstrated in the following sections present various mechanisms to allow one to traverse to source of data for this kind of values using the techniques illustrated in Chapter 4.
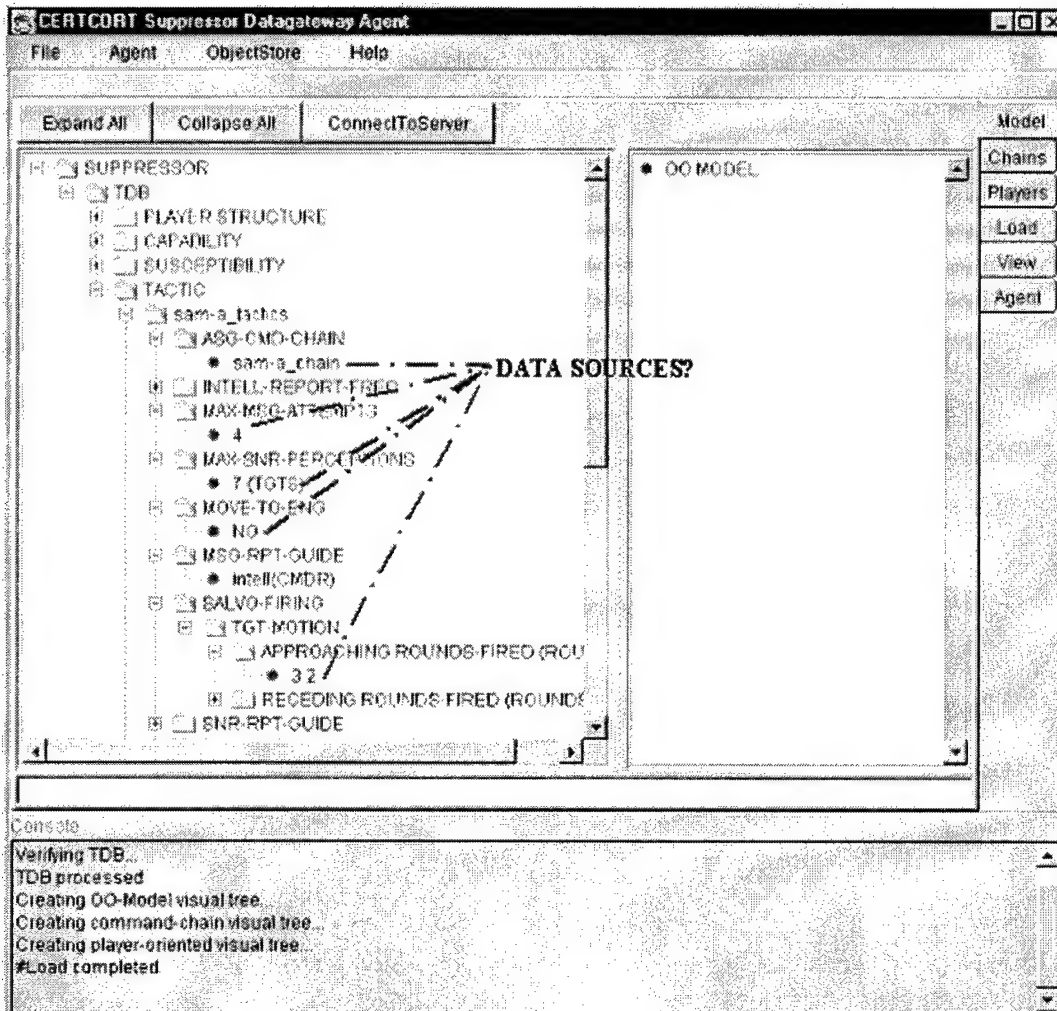
Figure 61- JTree representation of the Suppressor object.

The JTree representation in Figure 61 is the set up for all the applications in this research. The demonstration of the applications is presented in the next three sections.

5.4.1 Mirror Tree Model

In this model, the Suppressor Datagateway Agent's frame is split into two panes, the right pane has the mirror tree (see section 4.4.1) as in Figure 62.
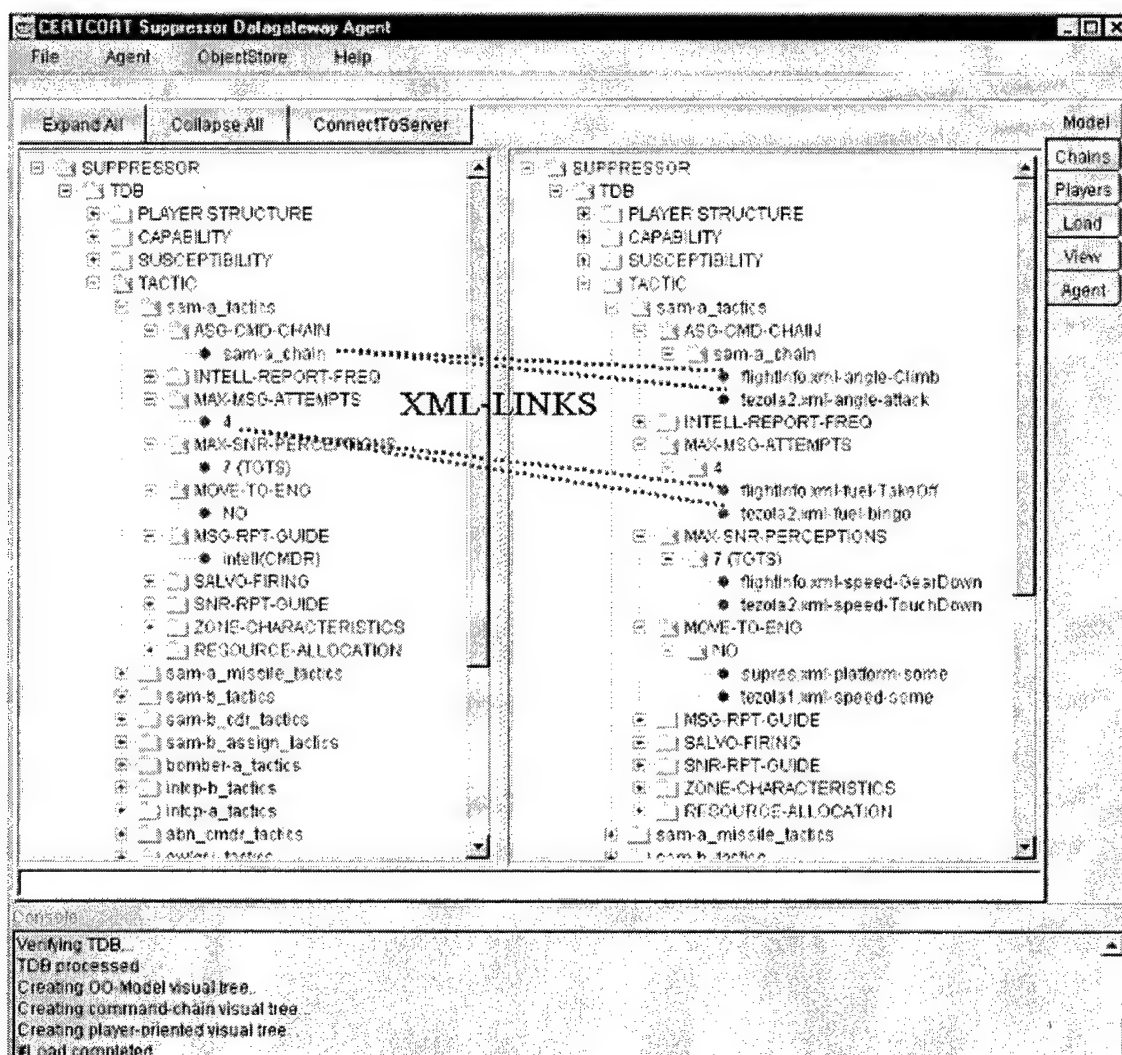
Figure 62- Mapping between the original JTree and the mirror JTree.

To create the JTree on the right pane in Figure 62, the TDB_M.txt file shown in

Figure 60 is selected. This "mirror" file which contains the xml-link values augmenting the

"approved" Suppressor grammar is parsed by a new parser (see section 4.4.1) which recognizes

the grammar in this file. The Suppressor object model populated by the mirror text is converted to

a JTree by the new *toTree_M ()* methods (described in section 4.4.1). For example, "Sam-

a_chain" has two xml-link values: "flightInfo-xml-angle-Climb" and "tezola2.xml-angle-attack."

This means that in order to modify this data item value, the scenario builder needs to view

fragments of two different XML documents. The mechanism to traverse to the XML documents

92

pointed by these xml-link values is activated by selecting the **Find-SplitTree** menu item in the

pop up menu shown in Figure 63.
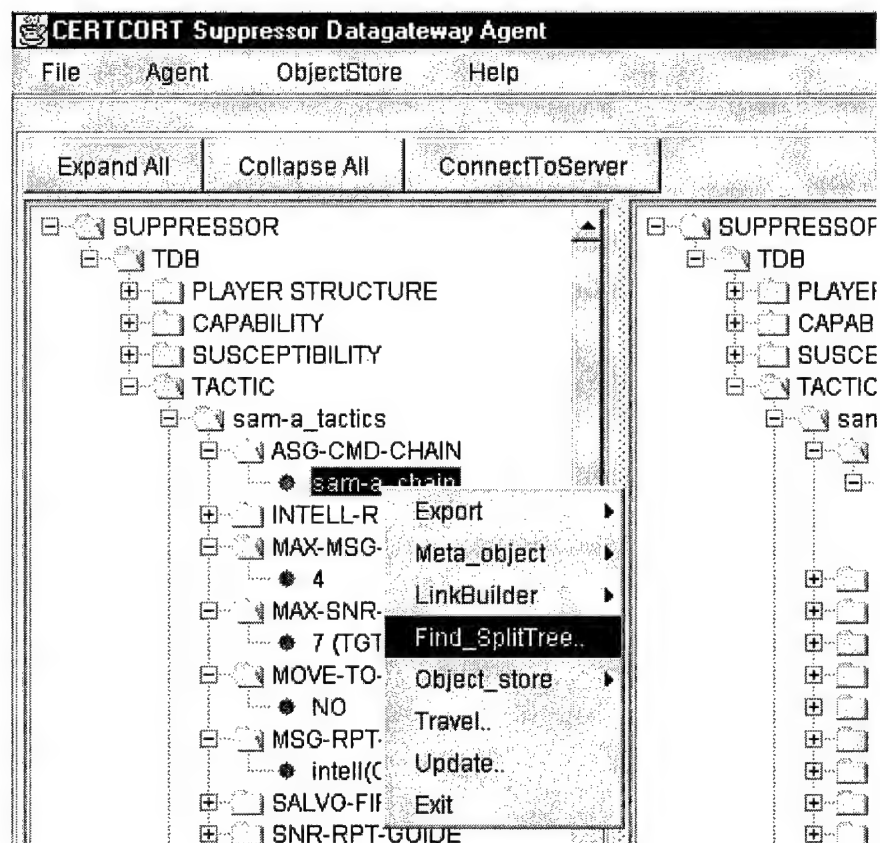
*Figure 63- The pop up menu to activate mechanisms.*

The selection of **Find_SplitTree** menu item invokes the mechanism of the mirror tree

model to find the xml-links, and parse the XML documents pointed. The data extracted from the

XML documents via the DOM API is presented to the user in display windows as shown in

Figure 64.
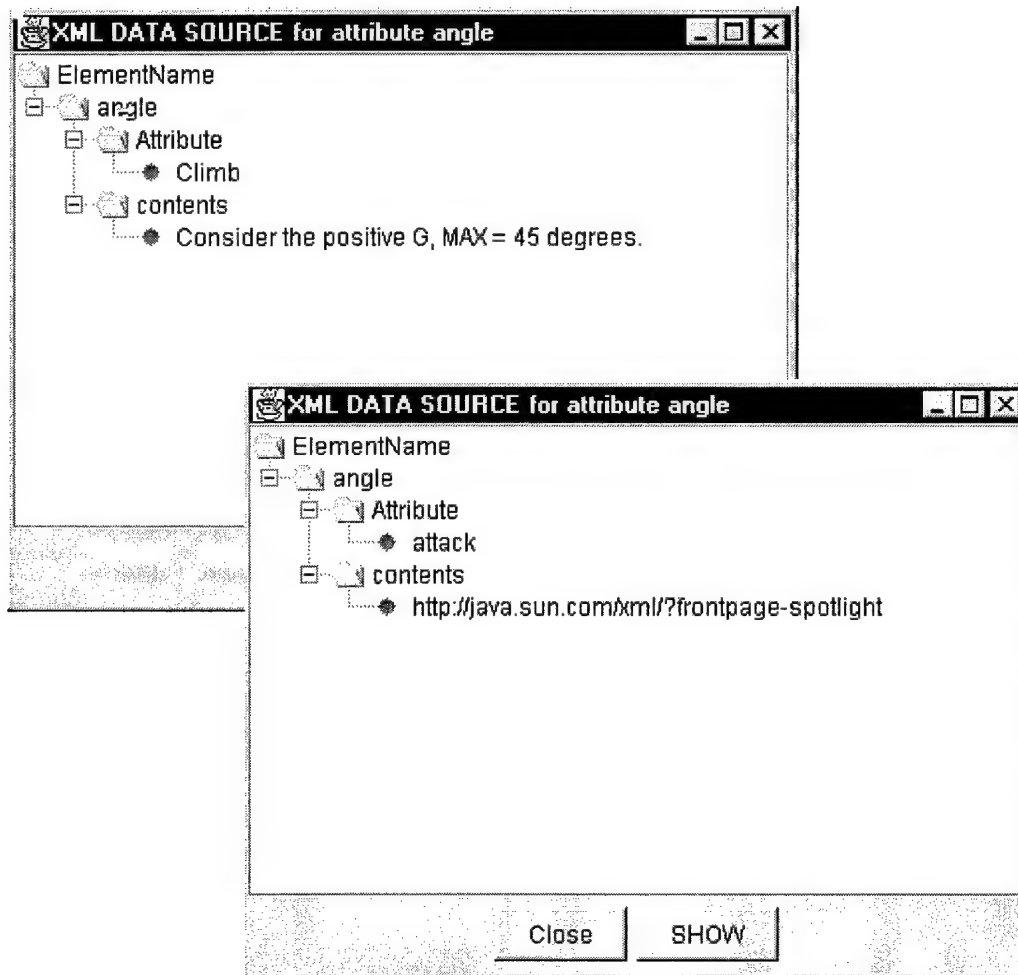
Figure 64- XML document fragments shown in display windows.

Since this value is mapped to two xml-links, two display windows help to visualize the

XML document fragments in a tree like fashion. The main part of these displays, which the

scenario builder is interested in, is the child node of the "contents" node. The content may be of

various types such as text. If it is an internet address (a web page), as in the lower display window

in Figure 64, then selecting the **show** button opens that web page as in Figure 65. A system call to

the browser is made by passing the web page address as the parameter to Java Runtime object.

The result is shown in Figure 65.
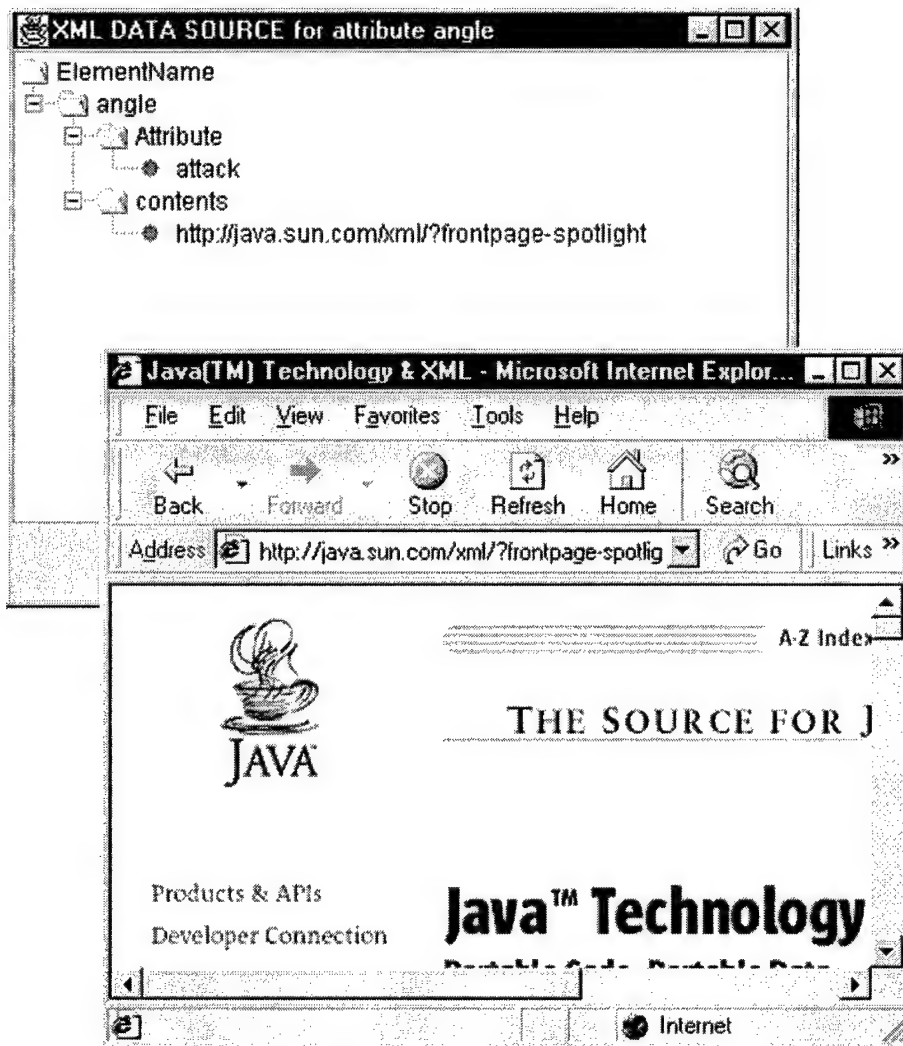
Figure 65- Traversing to a web page.

Figure 65 demonstrates the traversal from the display window representing XML document contents to a web page pointed by the content of the element of the XML document.

The contents of the element of the source XML document could point to an image file: in this case the **show** button is selected again and this invokes the objects which have the ability to process image files. This type of source XML element content and the result after invoking the image processing mechanism is shown in Figure 66.
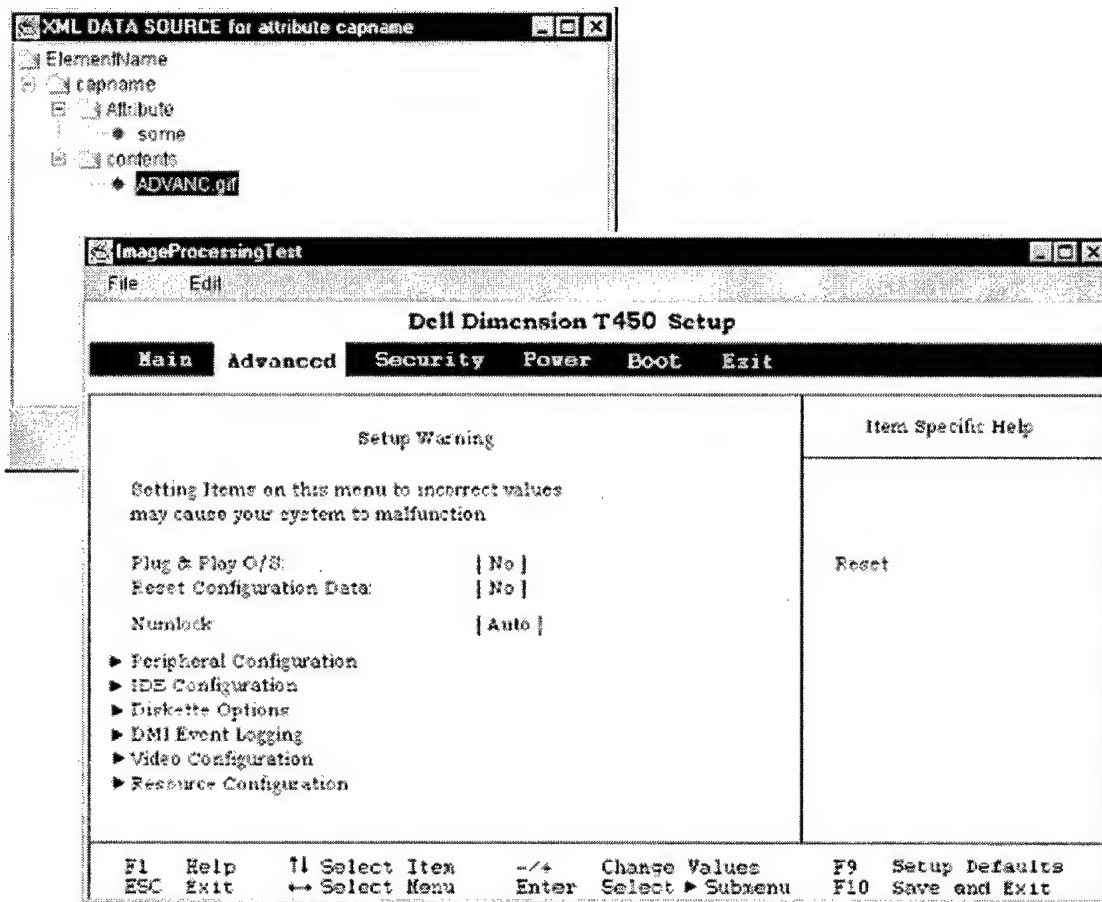
Figure 66- Processing an image file as the source of data.

5.4.2 Link Builder Model

When the scenario builder selects the **LinkBuilder** menu item on the pop up menu in Figure 63, the JTree representation of the original object model is sent to the process on another machine responsible for preparing the links over the network (see section 4.4.2). After receiving the JTree of the Suppressor object, the person in charge of preparing xml-links uses the pop up menu to invoke the required methods shown in Figure 67.
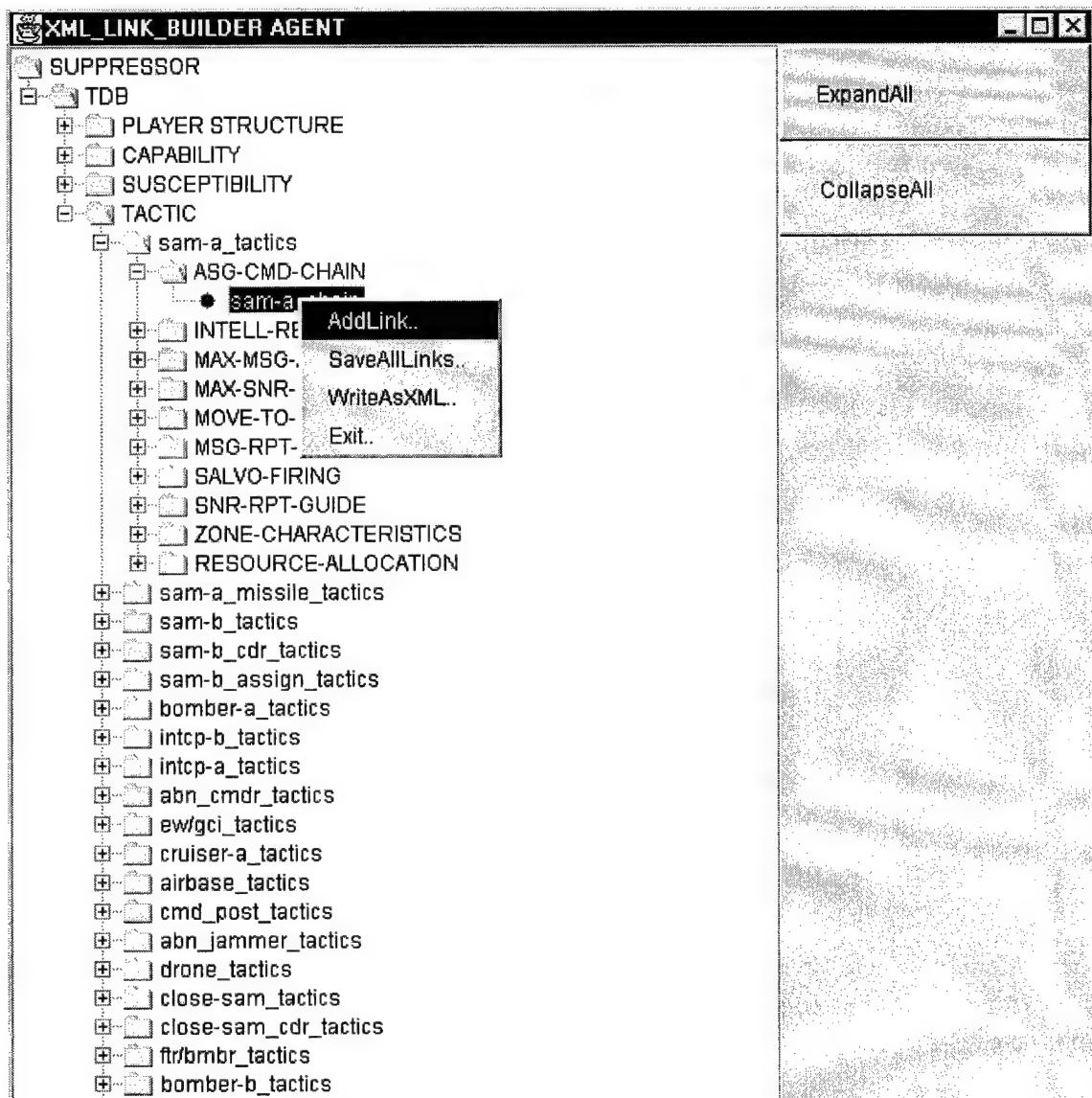
Figure 67- The Suppressor JTree on LinkBuilder side.

When the person in charge of preparing xml-links selects the **AddLink** menu item on the pop up menu in Figure 67, the dialog pane in Figure 68 appears. The user then supplies the field values in the dialog pane as shown. Every time link data is submitted through the dialog pane, it is mapped to a link object and this link object is added to a container. When the xml-links are all added, this container object is sent to the scenario builder over the network.
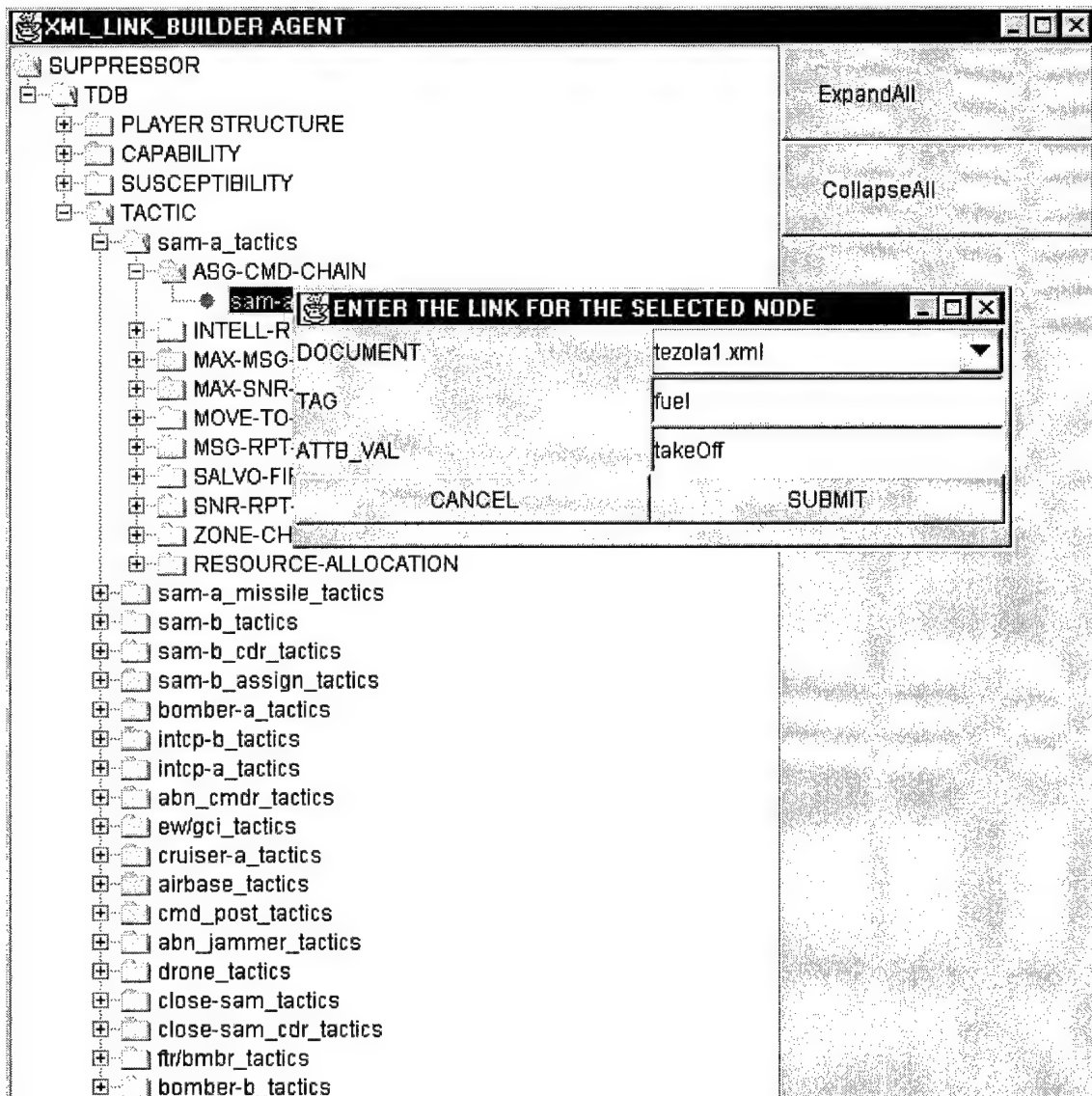
Figure 68- Adding xml-links for a selected node.

In addition to the field values in the dialog pane in Figure 68, the path of the selected node is also mapped to a link object. This is the main parameter to be able to have a matching between link objects and the nodes on the original JTree on the scenario builder side. After this point, the scenario builder receives the link objects container. The rest of the mechanism is the same as depicted in Figures 64, 65, and 66.

5.4.3 Meta-Class Instance Model

The scenario builder has the options given in Figure 69 to activate the meta class instance model. Selecting **FindMeta** on the pop up menu in Figure 69 invokes the methods to traverse the meta object and get the default xml-link values from the meta-class objects. The default xml-link values have only one argument (the name of the XML document): no specific tag is defined in the default xml-link values. So the mechanism builds a JTree representation of the DOM of the entire source XML document as in Figure 70.
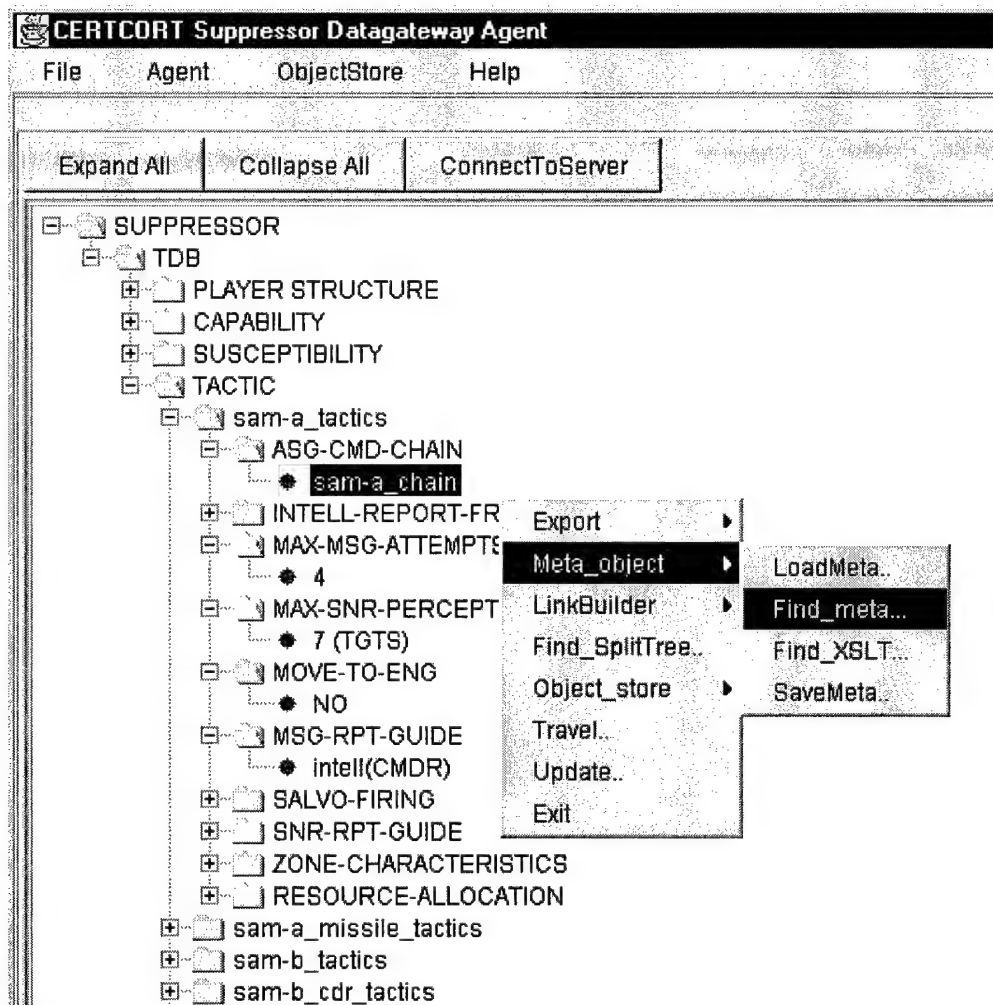

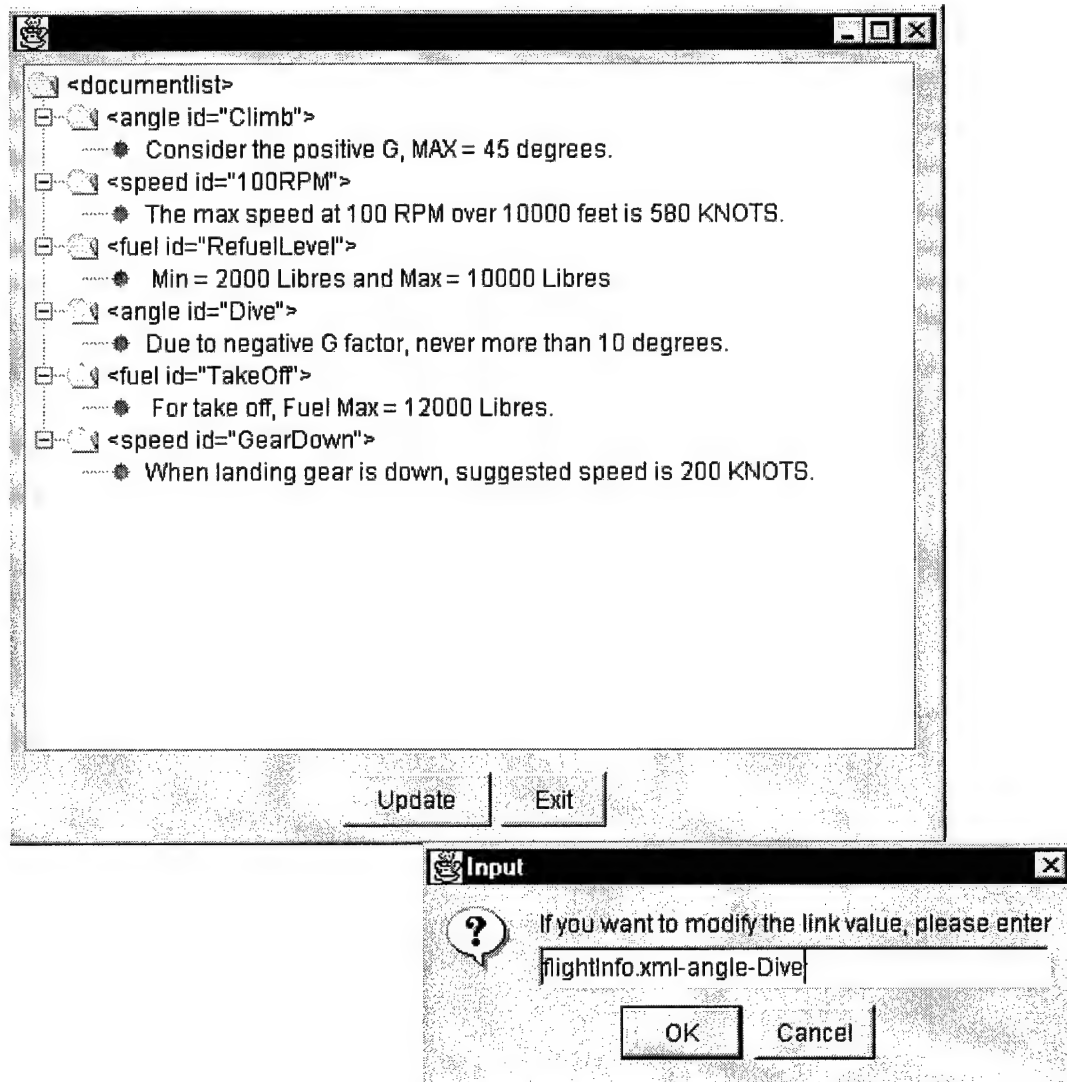*Figure 69- Submenu for Meta-object model.*

Figure 70- JTree of DOM of an XML document.

The JTree representation in Figure 70 helps the user to view the XML document in a hierarchical fashion. One important aspect of this is that if the scenario builder decides that there is no need to view the whole XML document, then he can update the xml-link value and make it more specific. The meta object model which has the modified xml-link values is written to a file persisted by the Java serialization mechanism so as not to lose the new xml-links. The **SaveMeta** option on the pop up menu in Figure 69 activates this mechanism. Selecting **LoadMeta** on the same menu restores the meta object model from the persisted file. When the user selects **Find-**

meta, the xml-link is not the default value but the last modified one. The result is more specific than in Figure 70, as shown in Figure 71.
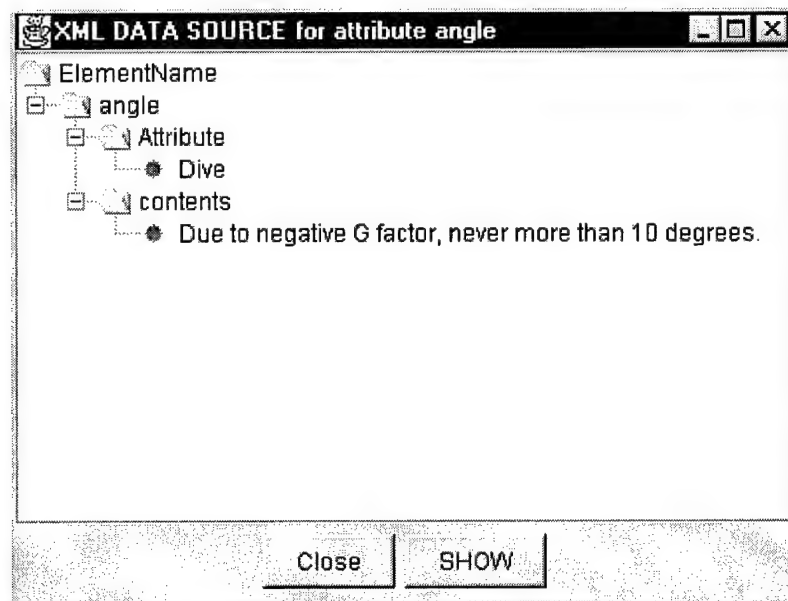
Figure 71- Result of making xml-links more specific.

As shown in Figure 71, if the previous scenario builder found the default link value too general, he changed and decided that there was no need to view and search the whole document. Considering that the XML document is too large, it is more efficient to specify exactly which part of the XML document is necessary to the scenario builder. Consequently, no longer the entire document, but the related portion of it, is returned to the user.

Another way to preserve the state of the meta object between runs is to store it in an OODBMS, such as eXcelon™ Corp.'s ObjectStore as discussed in Chapter 4. For this purpose, the scenario builder selects the **SaveToOs** menu item in the pop up menu depicted in Figure 72. This saves all the meta objects to ObjectStore. In the consecutive runs, the scenario builder can manipulate and interact with the meta object model stored in ObjectStore by selecting **FindLinkOS** menu item.
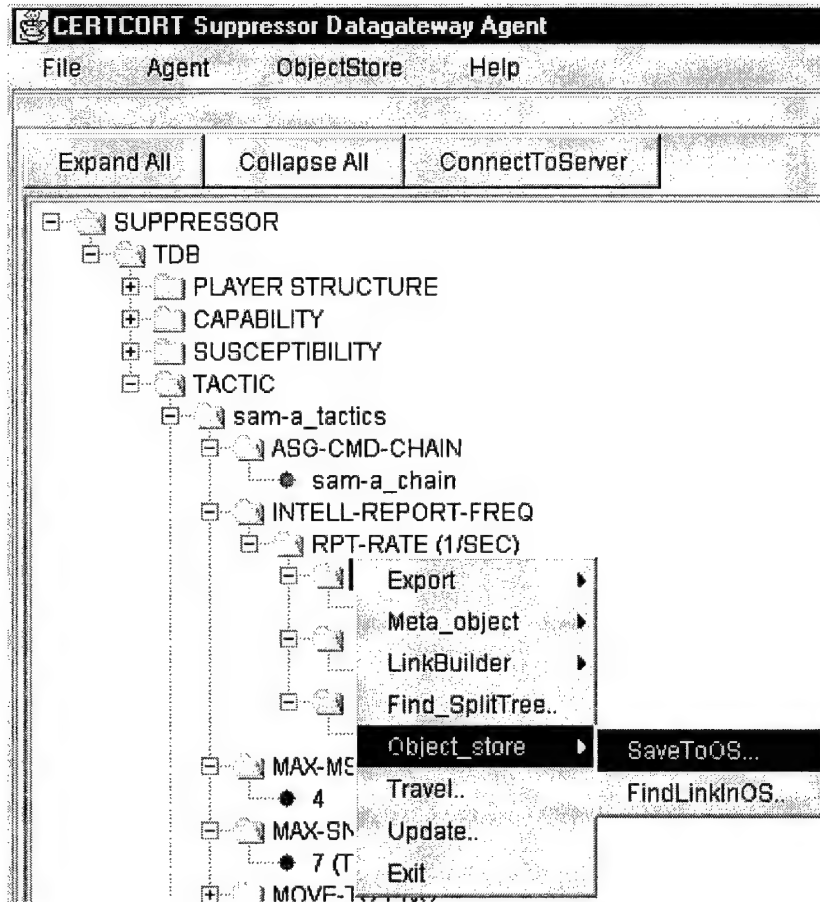
101

Figure 72- Interaction with the ObjectStore.

One last menu item from Figure 69 is **Find-XSLT**. Selecting this menu item invokes the mechanism which is proposed as an alternative to the DOM API for extracting data from XML documents. A meta object model also contains xslt-info values as described in section 4.4.3.2. So the first thing done is to find the matching xslt-info for the selected node. Then depending on the matching xslt-info, an XSL is created generically as illustrated in Figure 73.
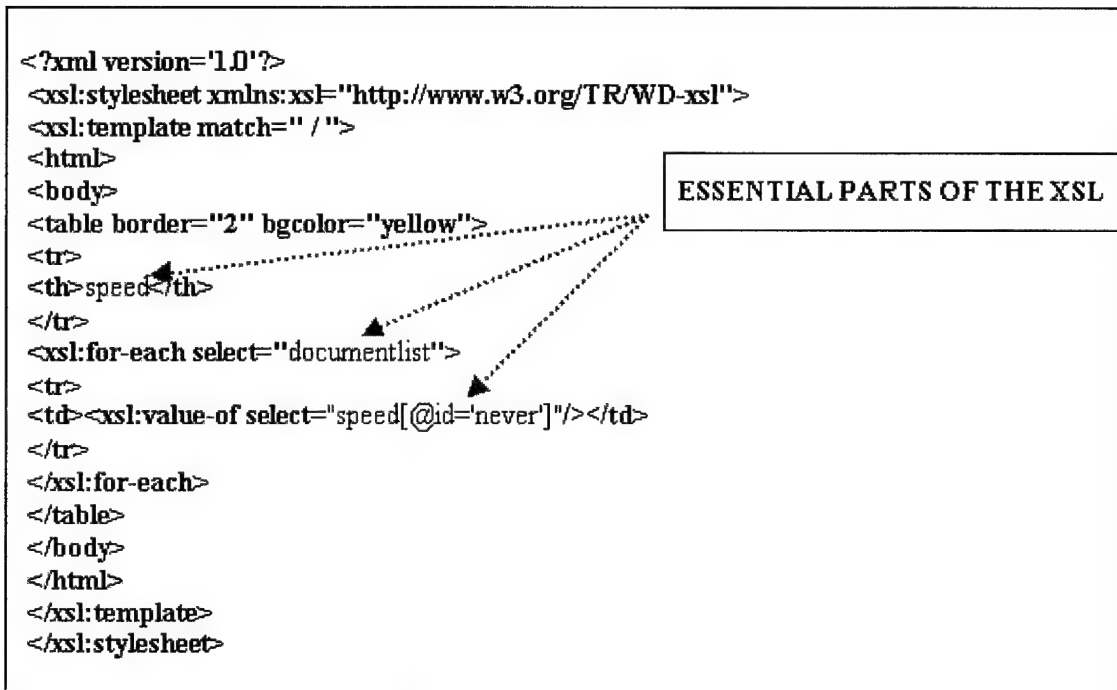
```
<?xml version='1.0'?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
<xsl:template match=" / ">
<html>
<body>
<table border="2" bgcolor="yellow">
<tr>
<th>speed</th>
</tr>
<xsl:for-each select="documentlist">
<tr>
<td><xsl:value-of select="speed[@id='never']"/></td>
</tr>
</xsl:for-each>
</table>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

ESSENTIAL PARTS OF THE XSL

Figure 73- Generically created XSL.

In Figure 73, the portions of the XSL document marked as "essential parts of the XSL" are the ones most vital to transform XML documents. These parts change in accordance with the data source to be extracted. The values for these parts are mapped from xslt-info stored in meta objects. A system call is made to the browser application within the current application to transform and view the source XML document with the generically created XSL. The XML document is styled and manipulated by the rules defined in the XSL. The XML document fragment specified by the Xpath is returned to the user. The result is illustrated in Figure 74.
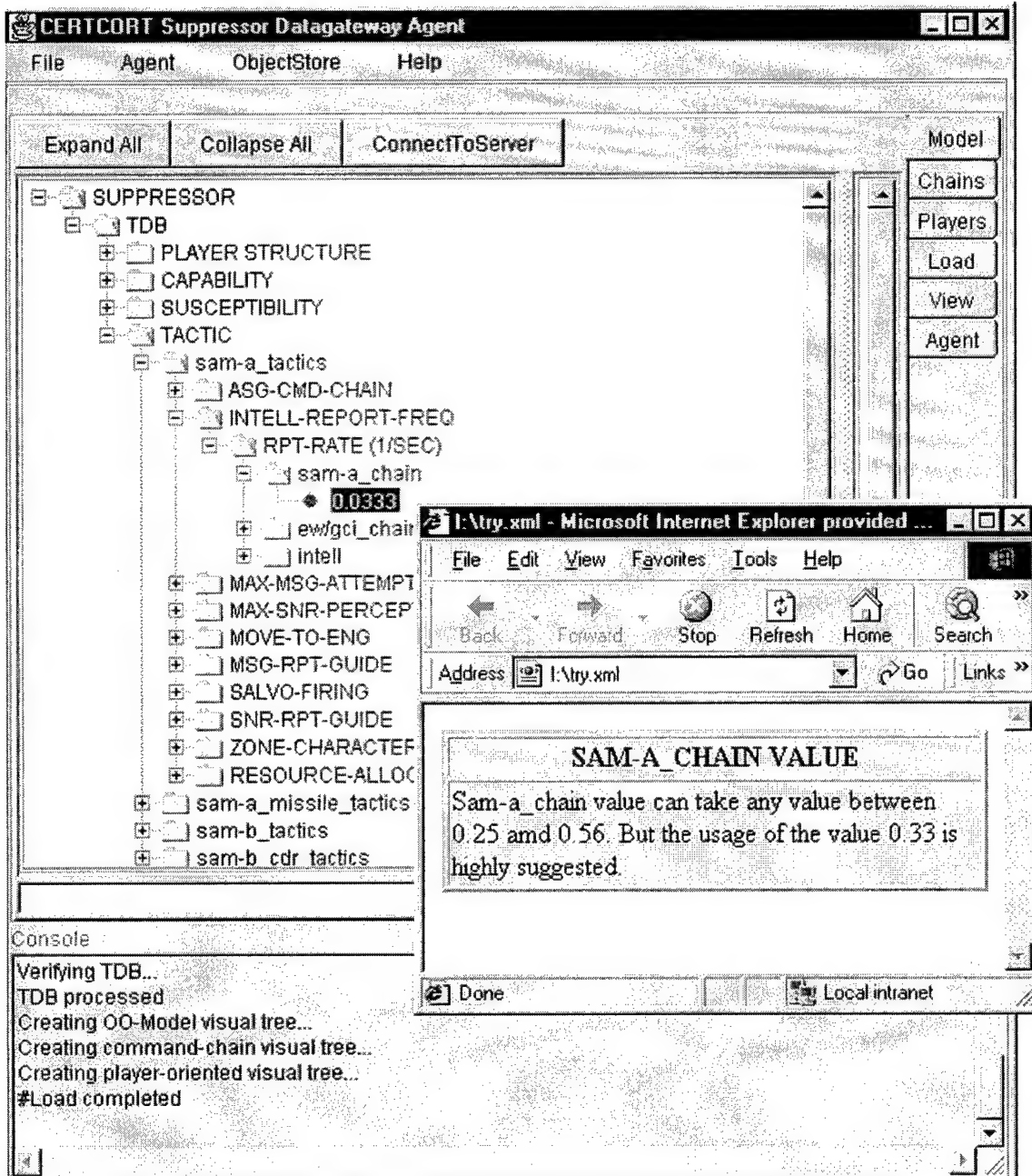
Figure 74- Result of using XSLT to extract data from XML documents.

5.5 CHAPTER SUMMARY

The demonstration for the three approaches developed in this work has been made in this

chapter. In the next chapter conclusions and future recommendations are presented.

6 CONCLUSIONS AND RECOMMENDATIONS

6.1 RESEARCH SUMMARY

The goal established for this research was to develop, explore, and demonstrate mechanisms for defining and tracing back to the origin of source data for the attributes (data items) of a Suppressor TDB file represented as objects. The starting point for the research was the Suppressor scenario object model developed by McDonald. McDonald's application lacks the trace back capability provided by this work.

McDonald's object model was analyzed to ascertain avenues for inserting a source-linking capability. Three methodologies were developed; a mirror tree (source link) model, a dynamic link builder model and the meta-class instance model.

The "mirror tree" model augments the original JTree display of TDB by adding xml-link values as the children of the original leaf nodes. Also, a traversal mechanism from the original JTree to the mirror JTree was established making use of the paths with the tree. This traversal mechanism finds the xml-links and then invokes the XML parser object to fetch the related fragments of the XML document to the user. The XML parser extracts data from the documents via the W3C DOM API.

The second methodology developed is the "link builder" model. This mechanism works within McDonald's paradigm of collaborating agents which were registered with a central agent. The link builder model of this present work involves a scenario builder which functions as the information (link) requestor and the link builder which serves as the information provider. The present work treats a scenario builder and a link builder as agents registered with a central broker. This methodology also requires an advanced understanding of the Java Serialization mechanism to effect persistence, and facility with distributed computing mechanisms in Java. (The implementation and the demonstration of the integration of the link builder model with the existing CERCORT agent framework is presented in Appendix A.)

105

The third methodology, the meta-class instance model, constructs a new object model reflecting the class hierarchy of the Suppressor object model. In the fields of the objects of this model, two basic types of information are stored: xml-link values (like hypertext) and style sheet transform (xslt-info.) A deterministic finite automata (DFA) has been developed to traverse the meta-class instance model and find the desired component in the meta-class instance model. Xml-link values fetch the XML document fragments via a DOM API. Making use of xslt-info values provides an alternative to DOM as an approach for fetching XML document fragments. In all the previous approaches DOM API was used. The xslt-info helps to create a generic XML style sheet (XSL), which then allows one to manipulate XML documents.

The scenario builder (using the GUI application created by McDonald and augmented as described here in,) is given the ability to edit the field values of the meta-class instance model. The state of the modified meta object is preserved in two ways between runs. The first one is using the Java serialization mechanism and the second one is by using an OODBMS such as eXcelon™ corporation's ObjectStore. Experimentation shows ObjectStore provides a more efficient way to make objects persistent.

6.2 BENEFITS OF THIS RESEARCH

Foremost, the applicability of a nascent technology, XML, to the simulation scenario reuse problem domain is validated in this research. So far, we have seen how XML/XSL can provide an elegant approach to dynamic scenario data source tracing. Additionally and practically, a capability which did not exist in the previous work has also been provided. Specifically, the value of an attribute in an object model representing a Suppressor Type Database (TDB) can be traced to source data. Provided that data files which serve as the source of data are represented in a common format using XML, the data sources are accessible using the W3C DOM API or XSL.

In addition to the **syntactic correctness** checks performed by the *parse()* methods in the Suppressor objects, traversing dynamically to the source document also allows the scenario builder to check the **semantic correctness** of the attribute values of a Suppressor object.

This research also demonstrates that if the data sources are made **homogeneous**, the application required to extract data from these data sources need not be complex. Every application, which has the ability to parse XML documents, can read in that data because the data is represented in XML, a universal standard.

This concept simultaneously brings up the idea of reusing legacy Suppressor scenario files. Modifying the field values of an existing Suppressor scenario in object form and persisting the modified object model as a new scenario file.

6.3 FUTURE RESEARCH RECOMMENDATIONS

Standards and specifications for XML and XSL are still evolving and change frequently. The capabilities provided by XML get more mature everyday. This promises that new versions of the XML specification can be applied to the scenario reuse problem domain to enhance the data source tracing capabilities described in this work. For example, when this research was being written, the new Java XML parser version of **Sun JAXP 1.1** was under review. It has **XSLT plugability** which is not available in the older version JAXP 1.0. Also DOM level 2 specification has been released. Another evolving standard is XML Query Language (XQL) which has a SQL like query structure as in Figure 75 [48].

```
WHERE <person>
        <name></> ELEMENT_AS $n
        <ssn> $ssn</>
      </> IN "www.a.b.c/data.xml",

        <taxpayer>
          <ssn> $ssn</>
          <income></> ELEMENT_AS $i
        </> IN "www.irs.gov/taxpayers.xml"
CONSTRUCT <result> $n $i </>
```

Figure 75- XQL samples.

The XML query in Figure 75 depicts the ability of XQL to integrate data from different XML sources; "data.xml" and "taxpayers.xml." It is now possible that these kinds of changes in the XML technology can be applied to the subject problem domain.

Another future research topic can be to make a toolbox which will serve as an efficient, all-in-one application for the scenario builder. This toolbox could have abilities like **copy** and **paste** the values of the XML data sources to the JTree representation. Or it could have the ability to **drag** subtrees from the JTree representation of old scenario files and **drop** them to new JTree representation. In the prototype model this was achieved successfully. This could be an efficient way of reusing old scenario files and editing new scenario files.

One last future research recommendation is to extend the **meta-class instance model** to create a corresponding meta-class object for every instance in the Suppressor object. This can be done by developing mechanisms to create meta objects generically when the Suppressor objects are being populated and make a generic mapping between two object models.

APPENDIX.A

INTEGRATING THE LINK BUILDER MODEL WITH THE AGENT-BASED FRAMEWORK

In this section, the effort is to introduce how the link builder model presented in section 3.4.2 is integrated with the agent-based framework developed by McDonald. The link builder model was originally developed as a client-server system where the communication was established over a socket connection using object output and input-streams. It was a standalone application. However considering the AFRL/SNZW's overall initiative for the development of a Collaborative Engineering Real-Time database CORrelation Tool (CERTCORT), it is important to demonstrate the link builder model can be integrated with the existing agent-based framework. As stated in [18]:

"The greatest benefit offered by agent orientation can be seen in its ability to abstract the massively complex nature of CERTCORT into more manageable pieces and clearly definable problem domains."

1. The Existing Agent-Based Framework Architecture

To create the agent-based framework, McDonald implements the Multi-Agent Systems Engineering (MASE) which consists of the basic steps like: goal derivation for the system, determining the roles in accordance with the goal, transforming from roles into agents, determining and designing the communication protocol (the conversation between agents), and finally the system design.

The agent-based framework uses the agentMOM multi-agent development library to build the multi-agent system. AgentMOM package consists of two abstract classes, Agent and Conversation, and three concrete classes MessageHandler, Message, and Sorry. AgentMOM's object model is depicted in Figure 76.
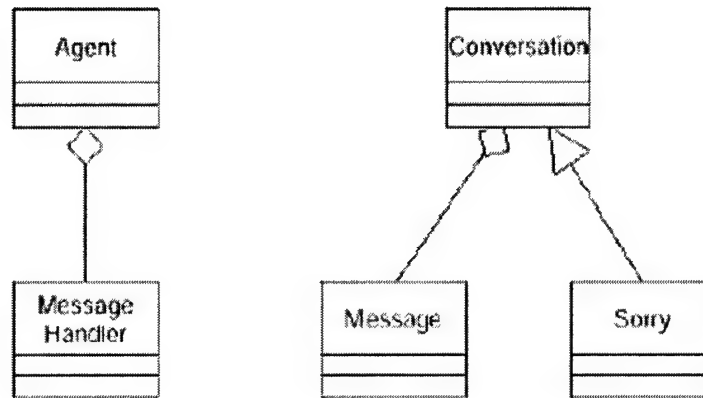
Figure 76- AgentMOM object model.

AgentMOM provides the basic building blocks for building agents, conversations between agents, and the messages that are passed in the conversations. AgentMOM implements communication via socket based messaging. AgentMOM thus requires the use of a dedicated port to listen for communications transmitted across a network of multiple agents. [50]

A message handler monitors the local port for messages while the conversation defines valid sequences of these messages. A separate Java thread is used for conversation, which is established by a socket connection with the other agent's message handler. The initial message is sent when a conversation thread is started and the message handler routes the message to an agent's *receiveMessage()* method. If a message is found in the list of allowable message types, then an agent knows a valid conversation can be started. A separate Java thread is used to establish the two sides of the conversation, which consists of other agent's response. The two conversation threads control all communication after the initial handshake, thus releasing the normal agent application to its event queue. [51]

A primary goal of this framework is information exchange from scenarios and data sources. For this purpose, there are three primary agent types; information requestor, information provider, and broker (middle-agent) agents. The CERTCORT multi-agent framework initially provides registration services for two particular agent types: the requestor and the provider. In this sense, both type of agents (requestor and provider) are "clients" while the broker is acting in

the role of "server." Whether a provider or a requestor, an agent provides the type of "service" it offers to the system when registering with the broker. If it is a requestor, the type of service indicates the information type the requestor is looking for.

The broker keeps a list of the agents providing particular type of information and also another list for the agents seeking particular type of information. When the broker detects a registered provider for the particular type of information a requestor is seeking, the broker notifies the requestor by sending the provider information (which includes service type provided, host name, and port number) to the requestor. After this point, the actual exchange of information is done subsequently between the information provider and the information requestor. The roles switch from client/server to requestor/provider. The information providers in the CERTCORT framework provide a copy of the entire object that encapsulates the information the provider is representing. The figure below illustrates the sequence of the events between the three types of agents.
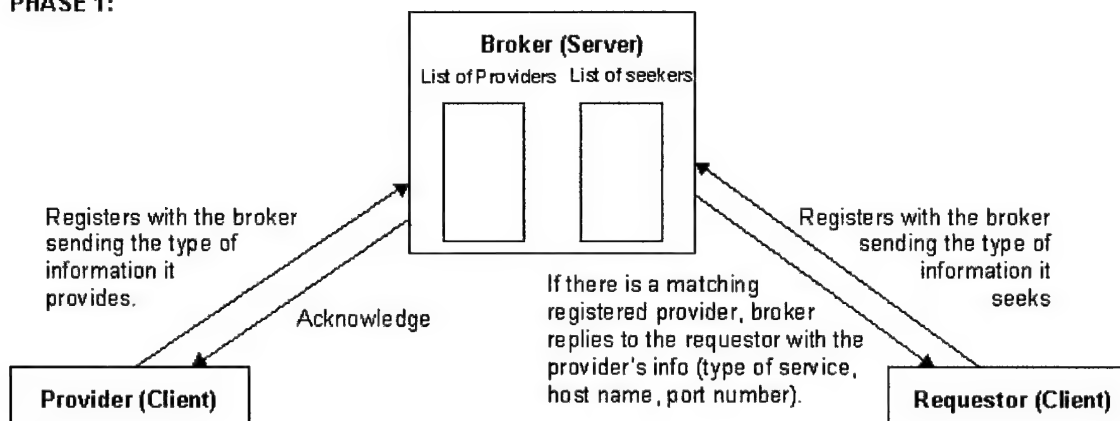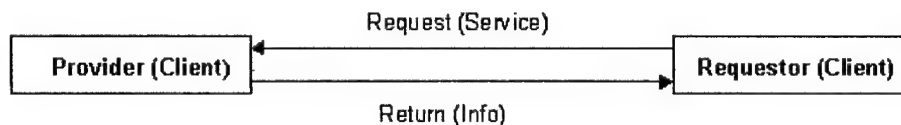
Figure 77 - Multi-agent framework architecture.

111

2. The Integration Process

By making use of the existing multi-agent framework, there is only need to create one additional agent, XML-link-builder agent, which is capable of providing the xml-link values desired. It registers with the broker in the same way as the other providers, sending the type of information it provides. The broker adds it to the list of providers.

The requestor used is Suppressor simulation builder agent and the broker is the CERTCORT broker provided by [18]. The sequence of the events is the same as depicted in Figure 77. Simulation builder registers with the broker for particular type of information (xml-links). Then the XML-link-builder agent registers with the broker advertising that it is capable of providing xml-links. When the CERTCORT broker detects a registered provider for the type of information the Simulation builder is seeking, it sends the provider info (type of information provided, host name, and port number) to the Simulation builder agent.

The rest of the communication takes place between the Simulation builder and the XML-link-builder. Simulation builder agent contacts the XML-link-builder to receive the type of information the XML-link builder agent represents. Finally, the XML-link-builder agent sends the copy of the object which encapsulates the desired xml-link values.

2.1 The Demonstration of the Integration Process

On the XML-link-builder agent side, xml-link values are created as depicted in Figure 78. The person sitting on this machine provides the link values through the dialog pane. The values provided (path from the selected node to the root of JTree, XML document name, tag name and attribute value) are mapped to a link object. Every time a link object is created, it is added to a container.
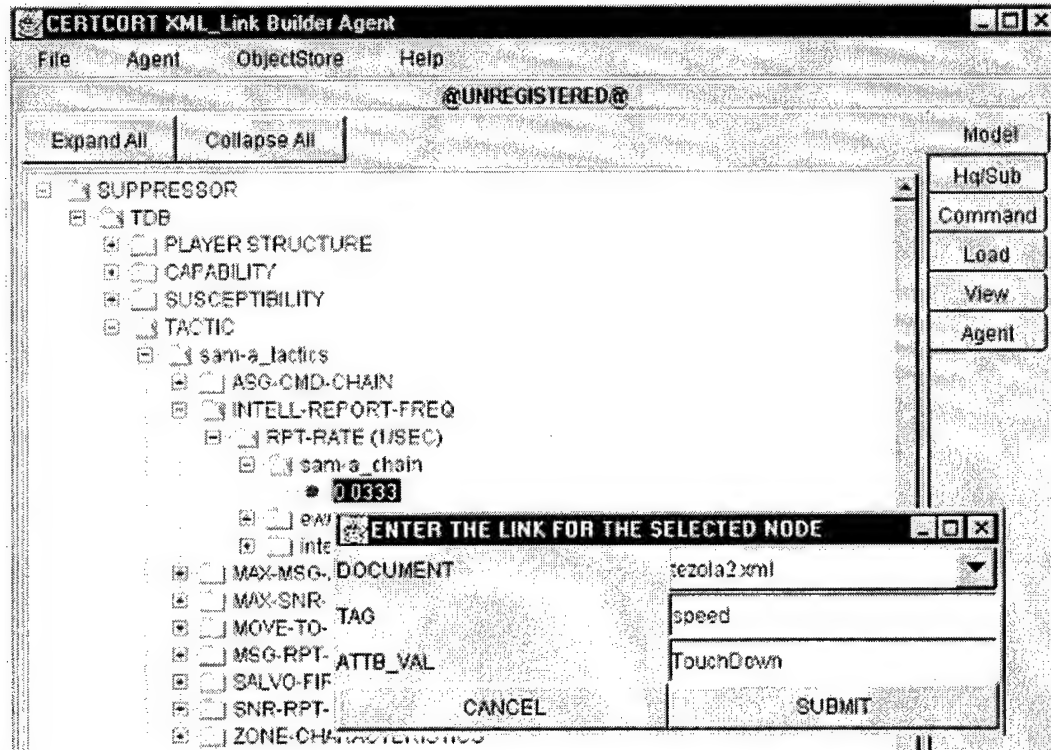
Figure 78- Creation of xml-links on the XML-link-builder agent side.

When all the link values are prepared, selection of the **AgentRegisration** menu item (see

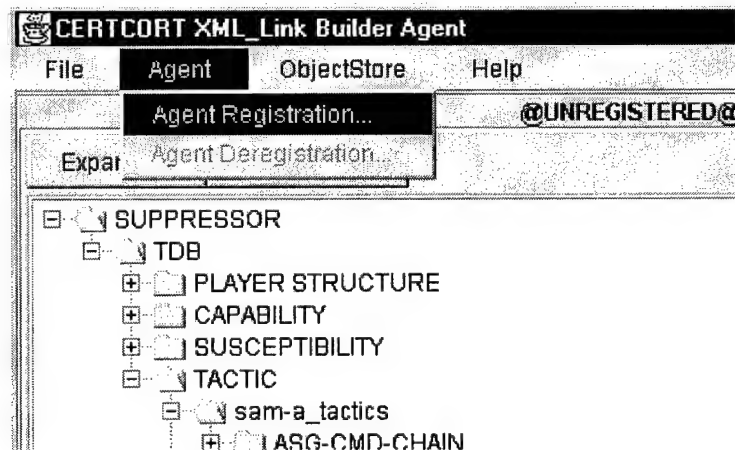Figure 79) registers this agent with the broker.

Figure 79- Registration of the XML-link-builder agent with the CERTCORT broker.

When registering with the CERTCORT broker, the link-builder agent sends the type of

service it provides. The broker keeps the list of registrars and the services they provide. The result

of the registration on broker side is shown in Figure 80.

113
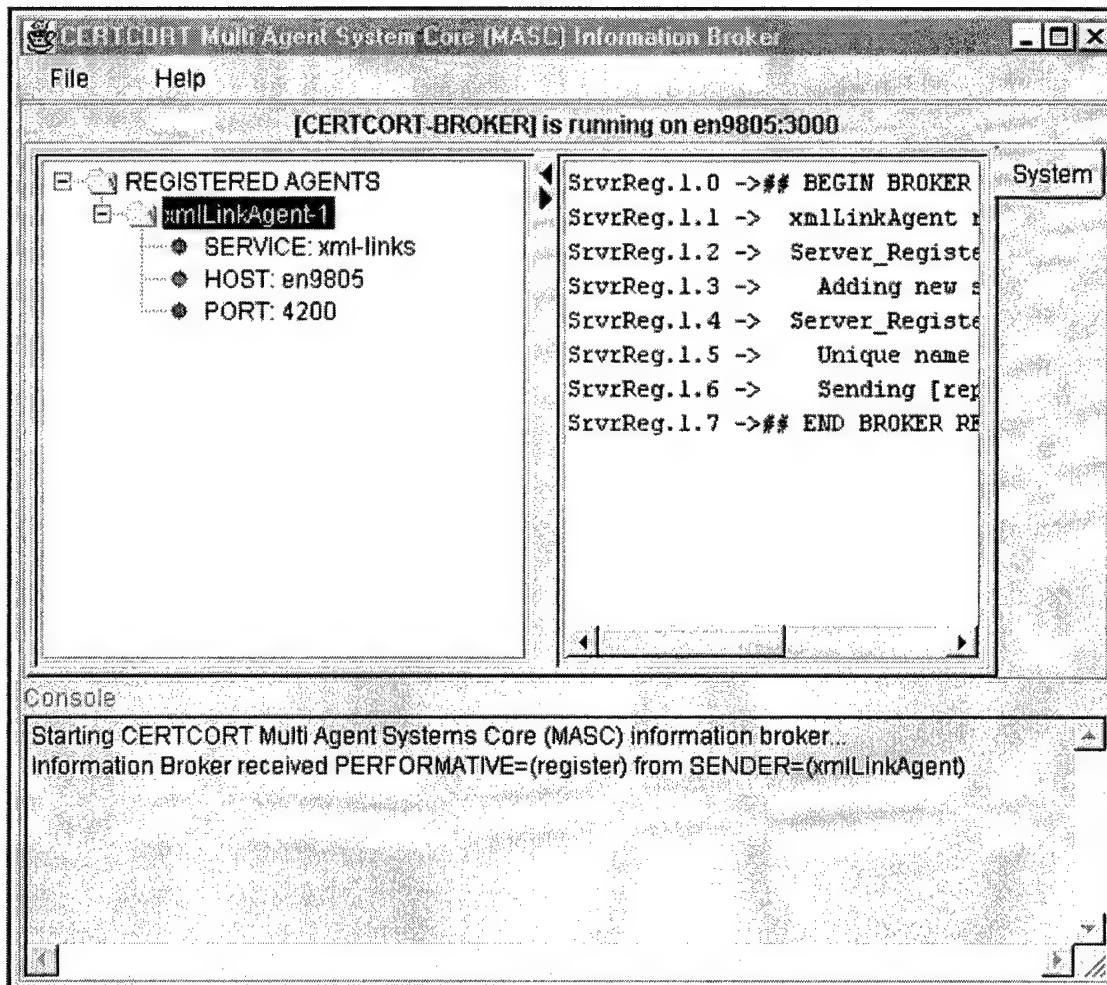
Figure 80- Result of registration on the broker side.

The CERTCORT broker and the XML-link-builder agents are run independently. But, because of the modifications made on the Suppressor Datagateway object by this author, there is a need to invoke the Suppressor simulation builder agent through the Suppressor Datagateway user interface. This is accomplished by selecting the menu item **StartInformationRequestingAgent** (see Figure 81.)
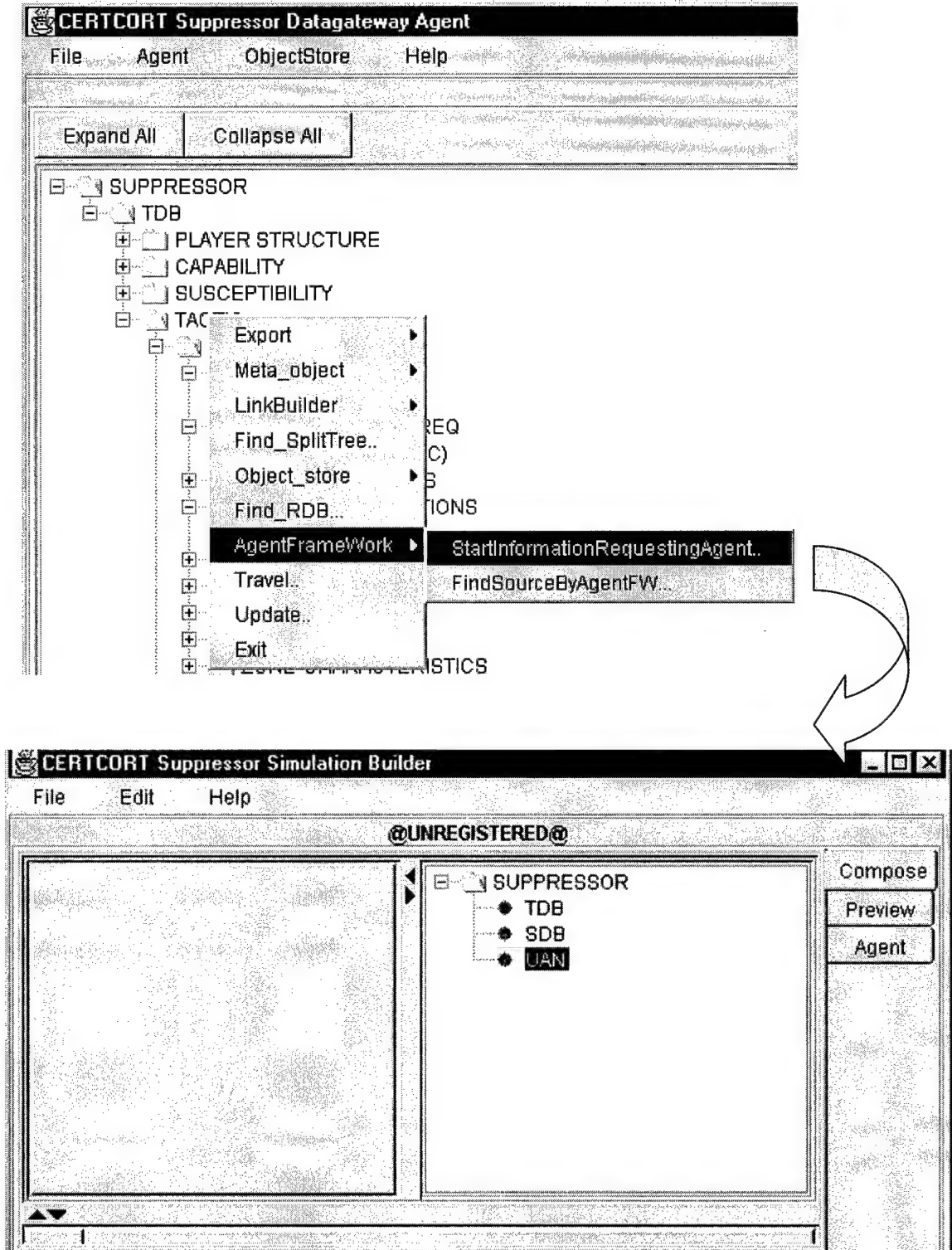
Figure 81- Invocation of the information requestor agent.

The information requestor, Suppressor simulation builder, registers with the CERTCORT broker

sending the type of service it seeks. The result of registration is illustrated in Figure 82.
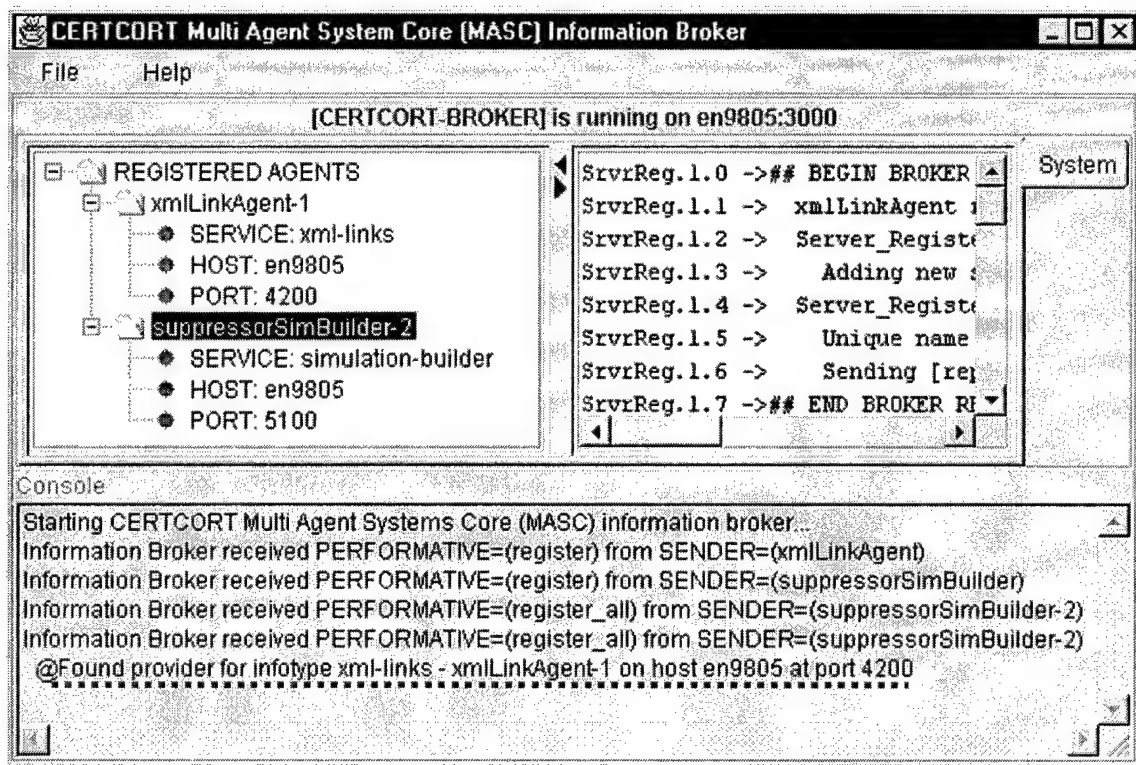

Figure 82- Registration of simulation builder with the CERTCORT broker.

Since the simulation builder asks for a particular type of service, "xml-links," when

registering with the broker, the broker checks its list of registered providers to see if there exists

an agent providing xml-links information. As marked in Figure 82 by the dashed line, the broker

detects the XML-link-builder agent on host "en9805", at port "4200." Consequently, the broker

sends the type of service provided, the host name, and the port number of the provider agent to
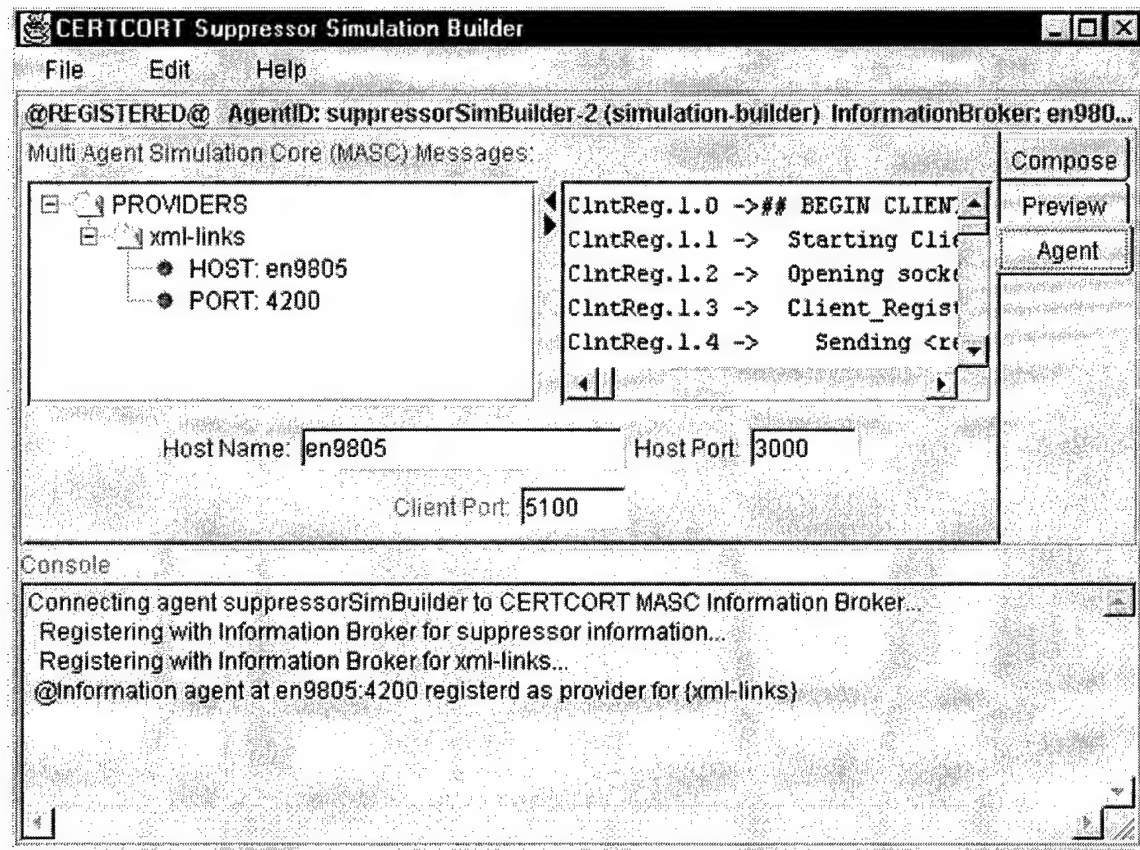
the simulation builder as depicted in Figure 83.

Figure 83- Simulation builder receives the provider's information from the broker.

After the provider agent's information are received from the broker, the simulation

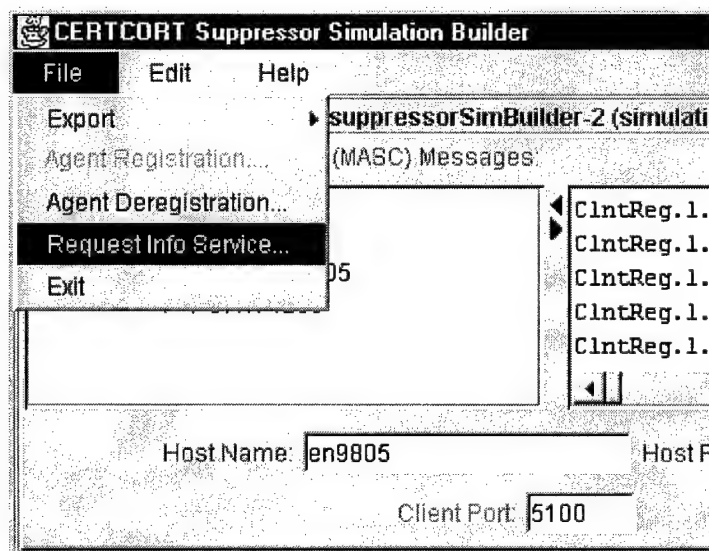builder contacts directly with the provider as in Figure 84.

Figure 84- Requesting information service from the XML-link-builder.

To request service; first, "xml-links" node on providers tree in Figure 83 is selected, then

RequestInfoService menu item shown in Figure 84 is selected. The XML-link-builder (provider)

sends the link-objects-container to the simulation builder (requestor). This mechanism is
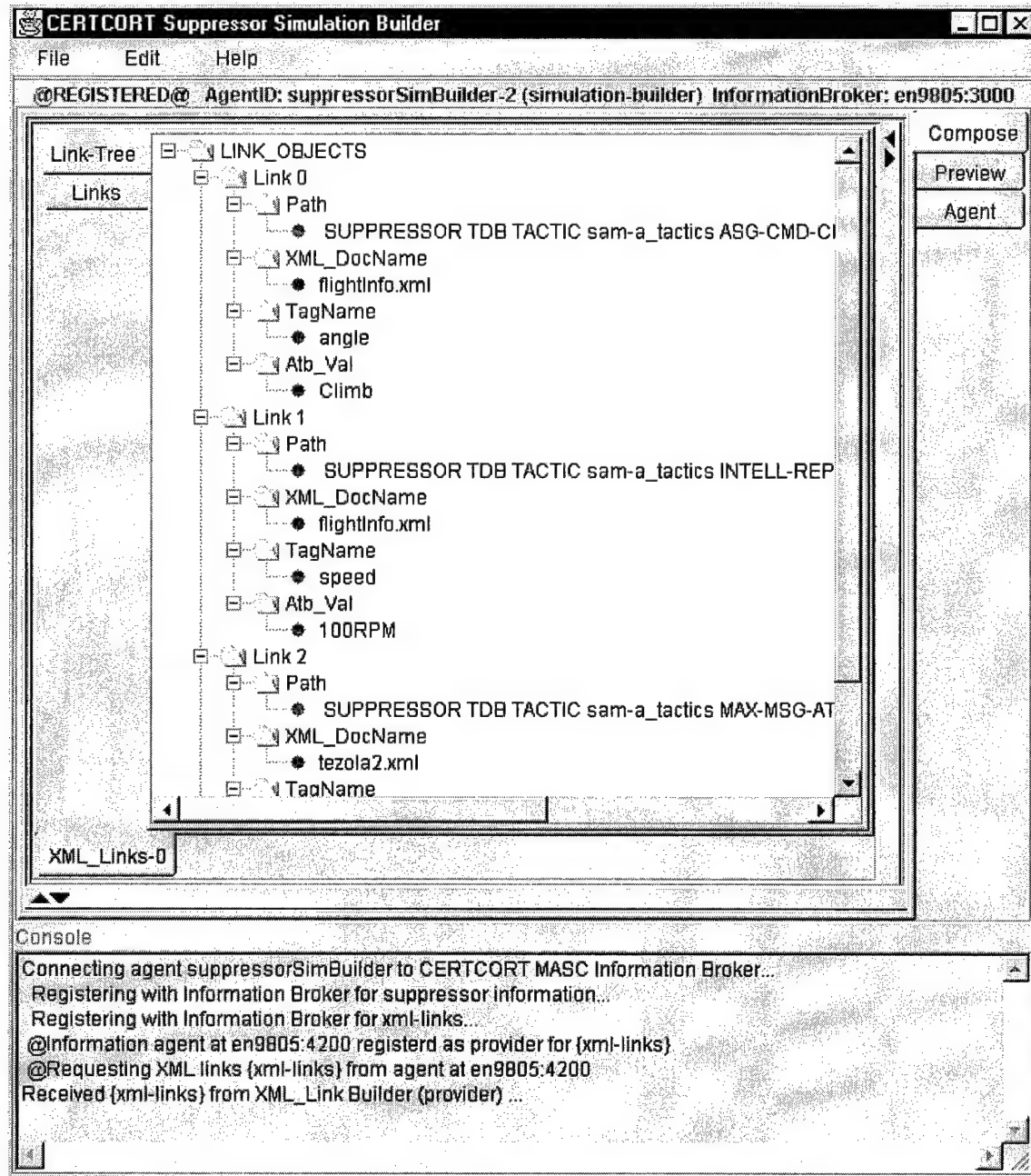
illustrated in Figure 85.

Figure 85- Simulation builder receives the link objects from the XML-link-builder.

The simulation builder has reference to its parent class (Suppressor Datagateway.) By this reference, the link-objects-container of the instance of the Suppressor Datagateway is also set by the vector received from the XML-link-builder. Thus, the user of the Suppressor Datagateway acquires the ability to trace back to source data by making use of the information provided by the multi-agent framework. To invoke the required mechanisms the user selects the **FindSourceByAgentFW** menu item shown in Figure 86.
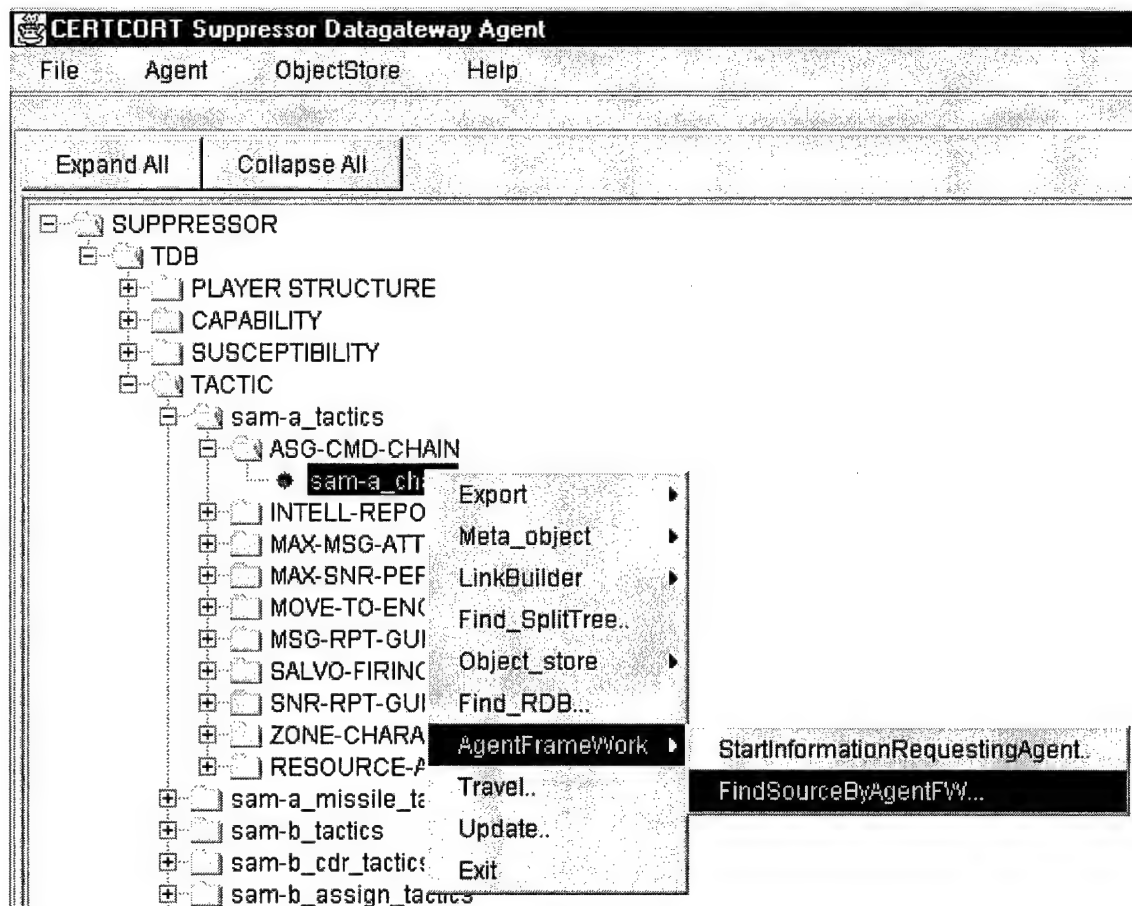

Figure 86- Tracing to the data source via the link information provided by the agent framework.

After the menu item selection demonstrated in Figure 86, the mechanism for finding the matching xml-link value in the links-object-container is the same as presented in section 4.4.2 and the mechanism to fetch XML document fragments in accordance with the matching xml-link value is the same as presented in section 4.4.1.2.

3. Conclusion

This study has shown that the models, particularly the link builder model, developed to trace back to data source for the attribute values of Suppressor scenario object can be integrated with the multi-agent system provided by [18].

APPENDIX.B

PERSISTENCE MECHANISMS FOR THE LINK BUILDER MODEL

1. Introduction

This study aims to solve a problem encountered in a previous work done by this author. The background for the previous work and the real problem are introduced in section 2. In section 3, the contemporary technologies which fit into this problem domain are discussed. The approach to solve the problem introduced in section 2.1 is presented in section 4. Finally, conclusions follow in section 5. The background for the previous work and the actual problem is introduced next.

2. Background

The previous work implemented three main models to find the source of data for the field values of an object model. The object model, which is called as "Suppressor" object model, is first populated by parsing a syntactically correct text file. A visual JTree representation of the Suppressor object model is created as depicted in Figure 87.
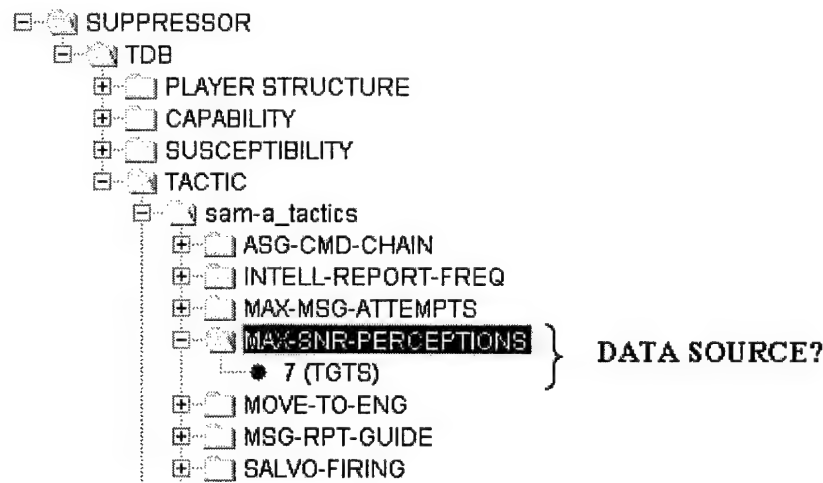

Figure 87- JTree representation of the Suppressor object model.

What is achieved in these three models developed is to traverse, for example, to the data source of the value "7 (TGTS)" of "MAX-SNR-PERCEPTIONS" as shown in Figure 87.

The link builder model, which is one of these three models developed in the previous work, forms the basis for this study. The figure below, depicts the mechanism of this model.
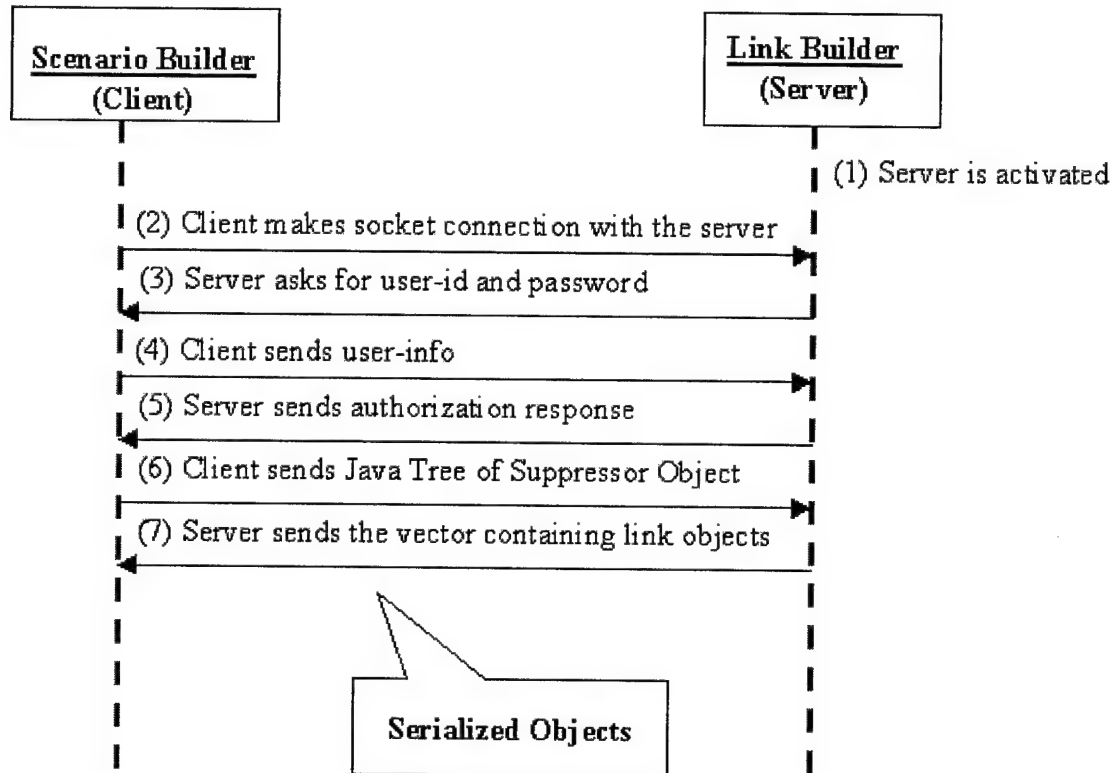

Figure 88- Sequence of events in the link builder model.

As shown in Figure 88, there are two parts taking place in this model. The scenario builder is responsible for the creation of the new scenarios, and to achieve this purpose he needs to traverse to the source documents or source document fragments. For this purpose he sends the JTree representation of the Suppressor object over the network to the link builder (server).

In the previous work, the data source documents were all converted to XML format. The parameter value, created by this author, passed to the XML parser in order to fetch XML document or its fragments is called **xml-link.** An xml-link has three main components. The XML document name (which determines the XML document to be parsed), the tag name of the related element (contributes to extract specific parts from the XML document), and the attribute value of the element (which is used to distinguish the elements having the same tag names).

122

The server has the capability and the authority to create xml-links for the attribute values of Suppressor object as illustrated in Figure 89.
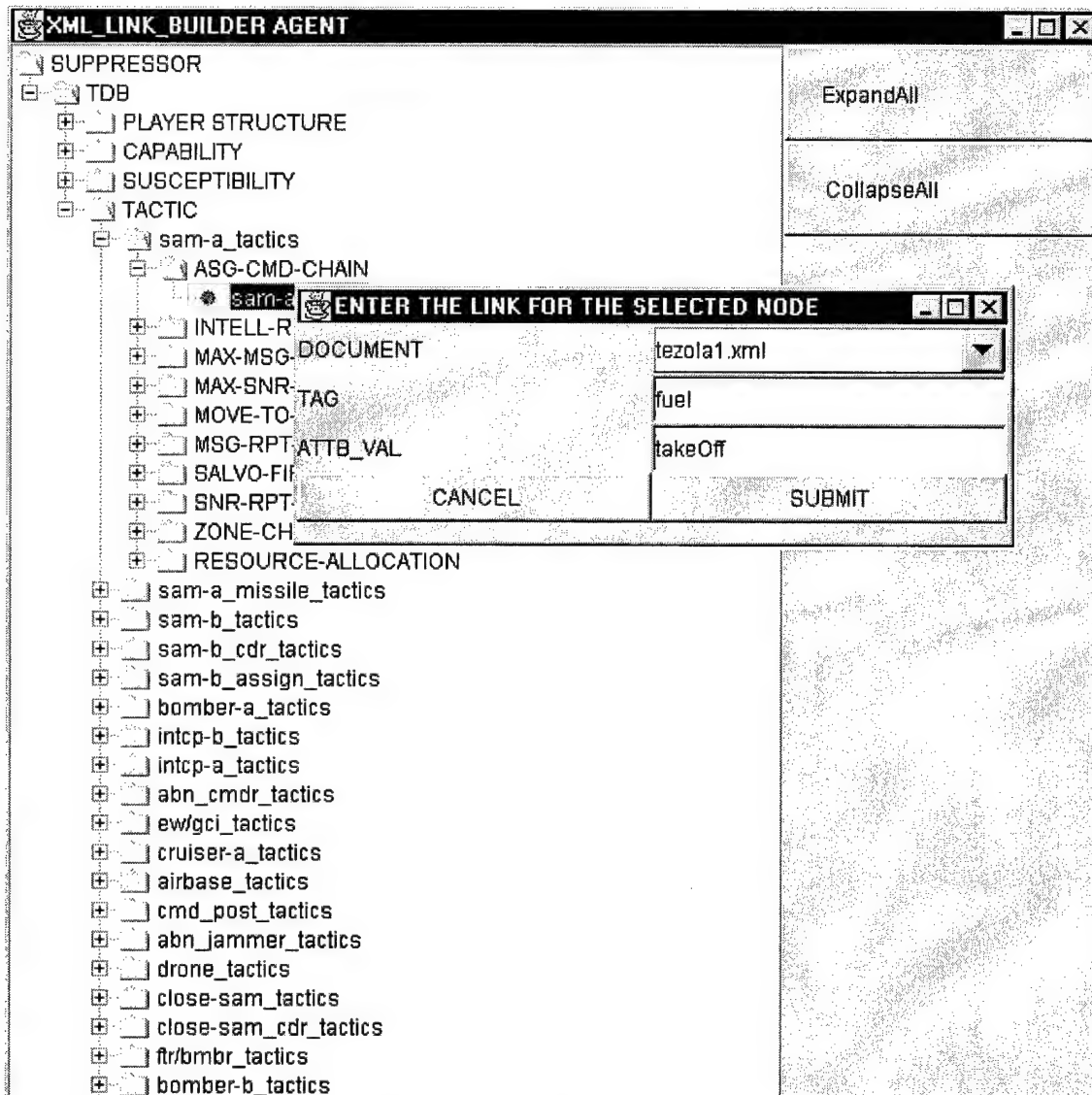

Figure 89- Creation of xml-links for the JTree representation of the Suppressor object.

As shown in Figure 89, when the link builder selects a node to create an xml-link for, a dialog pane appears. He is required to provide the document name, the tag name and attribute value through the fields in the dialog pane. These three values and additionally the elements of the **path array,** which consists of the nodes from the root of the JTree to the selected node, are mapped to a link object. There is a need to map the path array to the link object because it is the

main parameter which the scenario builder uses to find the matching link object. Every time an xml-link is created, a corresponding link object is created and this link object is put into a container. The problem which forms the subject of this work starts at this point, and it is introduced next.

2.1 Problem

In the previous work, the link-objects container is sent over the network to the scenario builder and the scenario builder can traverse to source XML documents via the data in the link-object-container. But there are two drawbacks associated with this link-builder model:

1- The xml-links prepared are lost when the application is terminated. There is a need to persist the created xml-links.

2- The scenario builder is in "wait state" until the link builder sends the link-objects container over the network to the scenario builder.

The theoretical discussion of the technologies which promise to solve these two problems is introduced next.

3. Contemporary Technologies to Persist an Object Model

This section introduces the contemporary technologies implemented in this work to persist an object model. The order of presentation is the Java Serialization mechanism, the Relational Database Management Systems (RDBMS), and the Object Oriented Database Management Systems (OODBMS), respectively.

3.1 The Java Serialization Mechanism

Serialization is the process of converting in-memory representation of an object into stream of bytes that allows it to be written out to a file. The object written out is stored persistently. When needed, the original object can be restored by reading the object data kept in the file.

124

Serialization is effective and efficient, as long as the number of objects to be serialized is small, because serialization has to read and write the entire object graph at a time. Serialization is also a drawback, when there is a need to update objects frequently. Dynamic queries can not be done on the files which have the serialized form of the object graphs. In addition, serialization does not provide reliable object storage. If the application crashes when the objects are being written out to a file by serialization, the contents of the file are lost.

3.2 Relational Database Management Systems (RDBMS)

Relational database systems have been around for many years to persist and manage large amounts of data and are accepted standard technology for creating back-office data stores [52]. They use two-dimensional tables as the basic structure for managing large amounts of data and offer efficient techniques for data storage and retrieval. The basic features of RDBMS are [53]:

1. Data abstraction: Users deal with conceptual representation of the data that includes little control over how the data is stored.

2. Basic database architecture: Software layers hide the low level details from the user.

3. Support for multiple users: Sharing data, concurrency, transaction processing, multiple data views.

4. Support for various types of user: Database administrators, database designers, end users.

5. Multiple ways of interfacing to the system: Query languages, programming languages, forms and command codes and menu driven.

6. Controlling information redundancy.

7. Restricting unauthorized access (database security).

8. Enforcing integrity constraints.

9. Backup and recovery.

Although some alternative database systems have been developed, the majority of the computing data across the world is still stored in RDBMSs. Relational databases are very good at storing, searching and retrieving data.

3.3 Object Oriented Database Management Systems (OODBMS)

Object oriented databases came into existence to offer a data model alternative to the one used by relational databases for persisting data in the 1990s. The need for a persistence model of this type arose with the advent of object-oriented programming languages. OODBMSs are a natural fit for persisting objects in OO world. The goal of object-oriented databases is to maintain a direct correspondence between real world and database objects. [52]

Object oriented database systems are good for manipulating data since they store it in object form, preserving the relationships between objects. An OODBMS works under the assumption that databases should store data objects that map directly to the objects defined in the programming language used for writing the application. These systems are based on object models that preserve the hierarchical relationships between objects, and offer a more sophisticated translation from objects to stored data.

Because relational databases store data as two-dimensional tables, they are not ideally suited for data manipulation, since data itself may be expressed in the form of complex structures. In object oriented languages data manipulation is much more efficient, expressing data as objects with rich features – including inheritance, polymorphism and encapsulation – for maintaining relationships between objects. When the data from an object oriented world is persisted in a two-dimensional table format, hierarchical relationships between objects are not preserved, making persistence and recovery of data a complex task.

ObjectStore, also, provides the mechanisms to overcome the disadvantages of serialization listed above. In fact, the steps to make objects persistent show some similarities to serialization. As with serialization, ObjectStore also makes use of **"readObject"** and

"writeObject" methods, which are automatically generated for each persistence capable class. But the superiority of Object Store over serialization is that it has the reliability of database management system (DBMS). This provides reliable object management and integrity. In addition to that, accessing and manipulating the persistent objects have in-memory-like performance [45]. It has a better performance for accessing large number of objects. It is possible to extract information from the persistent object graph by executing queries. Relational Database Management Systems (RDBMS) also provide robust and reliable data storage, but there is overhead with mapping Java objects into relational database tables and writing extra code to implement this mapping.

4. Approach

In this section, the approaches to solve the problem, introduced in section 2.1, are introduced. There are three main approaches in accordance with the contemporary persistence technologies introduced in section 3. How these technologies can be applied to the present problem domain is presented starting with the Java Serialization mechanism.

4.1 Persistence by the Java Serialization Mechanism

After the link builder receives the JTree representation of the Suppressor object from the scenario builder and the xml-links are created for the leaf node values (which represent the attribute values in the original object graph) on the JTree, the xml-links are mapped to the link objects and these link objects are put in a container. Then this container is sent to the scenario builder over the network. As discussed in section 2.1, this has two drawbacks: The contents of the container is lost after the application is terminated and the scenario builder has to be in "wait state" until the link objects container (the reply) is sent over the network by the link builder.

By using the Java serialization mechanism, it is possible to persist the link objects container. The in-memory representation of the container is written out to a file. Later, when the

scenario builder asks for the xml-links for the specific Suppressor object, the container is restored by reading in the file.

Regarding the number of objects to be serialized, serialization fits into this problem as introduced in section 3.1. However, the scenario builder still has to be in wait state until the container is restored from the file and sent over the network. Additionally, there is no way to make dynamic queries or make updates on the files which have the serialized form of the container object.

A more efficient way, persistence by RDBMS is introduced next.

## 4.2 Persistence by Relational Database Management System (RDBMS)

In this approach, the link objects container is persisted by making use of Relational Database Management System (RDBMS). Two-dimensional tables are the basic structure of RDBMS. So there is a need to map the link objects in the container to relational database tables. In this study, an indirect way of saving the link objects to the RDBMS is chosen as depicted in Figure 90.
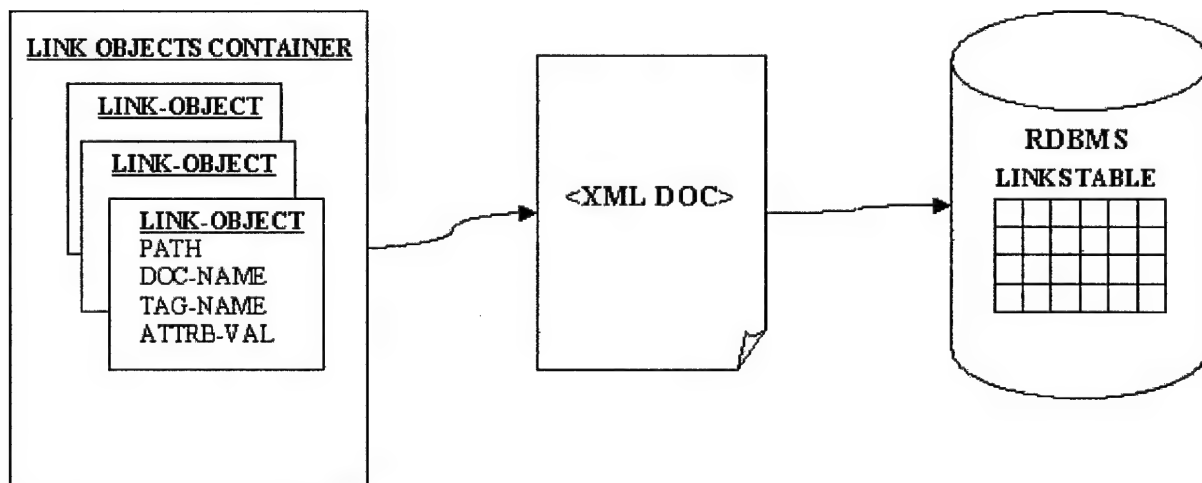
Figure 90- Persistence of link objects in RDBMS.

In fact the link objects could easily be mapped directly to relational database tables without having the intermediate step (converting link objects to XML format) shown in Figure 90. This intermediate step is applied in this persistence mechanism to demonstrate the conversion from an object model to XML document, and after that the XML document's content is mapped to relational database tables.

So, the first thing is to write the state of the link object container as an XML document, by tagging the data in the objects. In accordance with the hierarchy in the XML document, a **links** table is created in the RDB. The contents of the XML document is extracted by using Java DOM API and with this data the links table is populated. This mechanism is depicted in Figure 91.
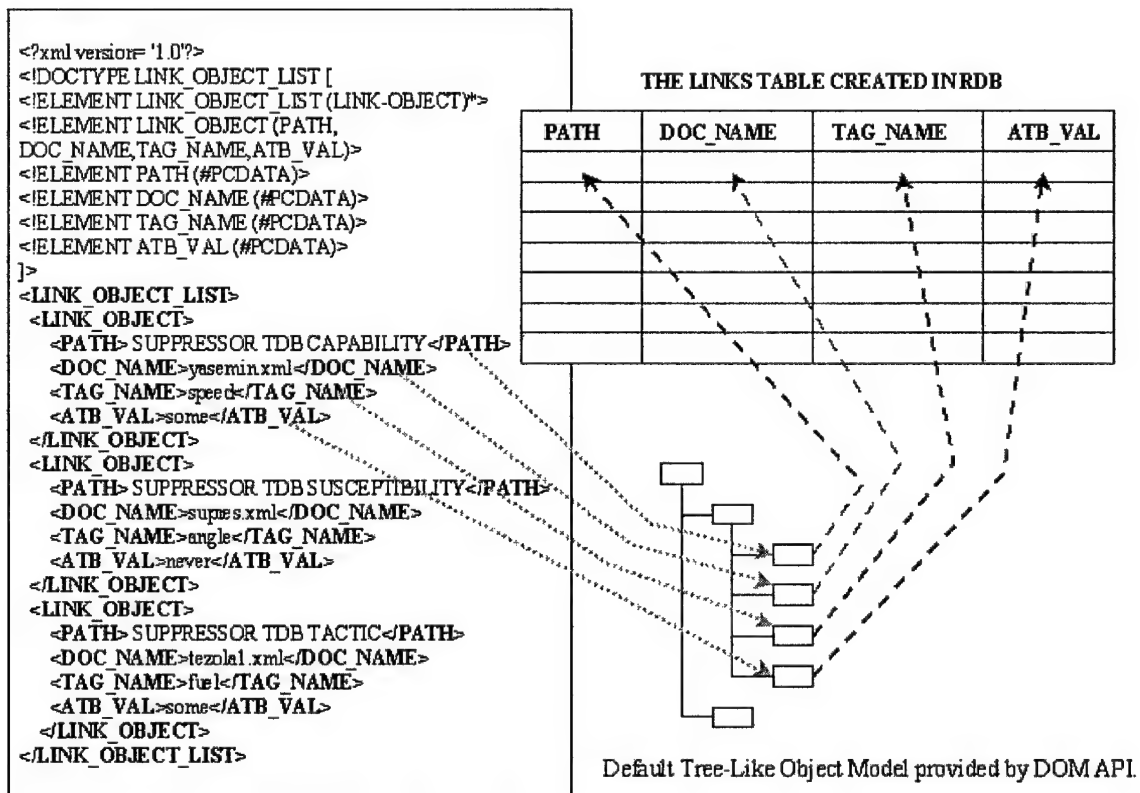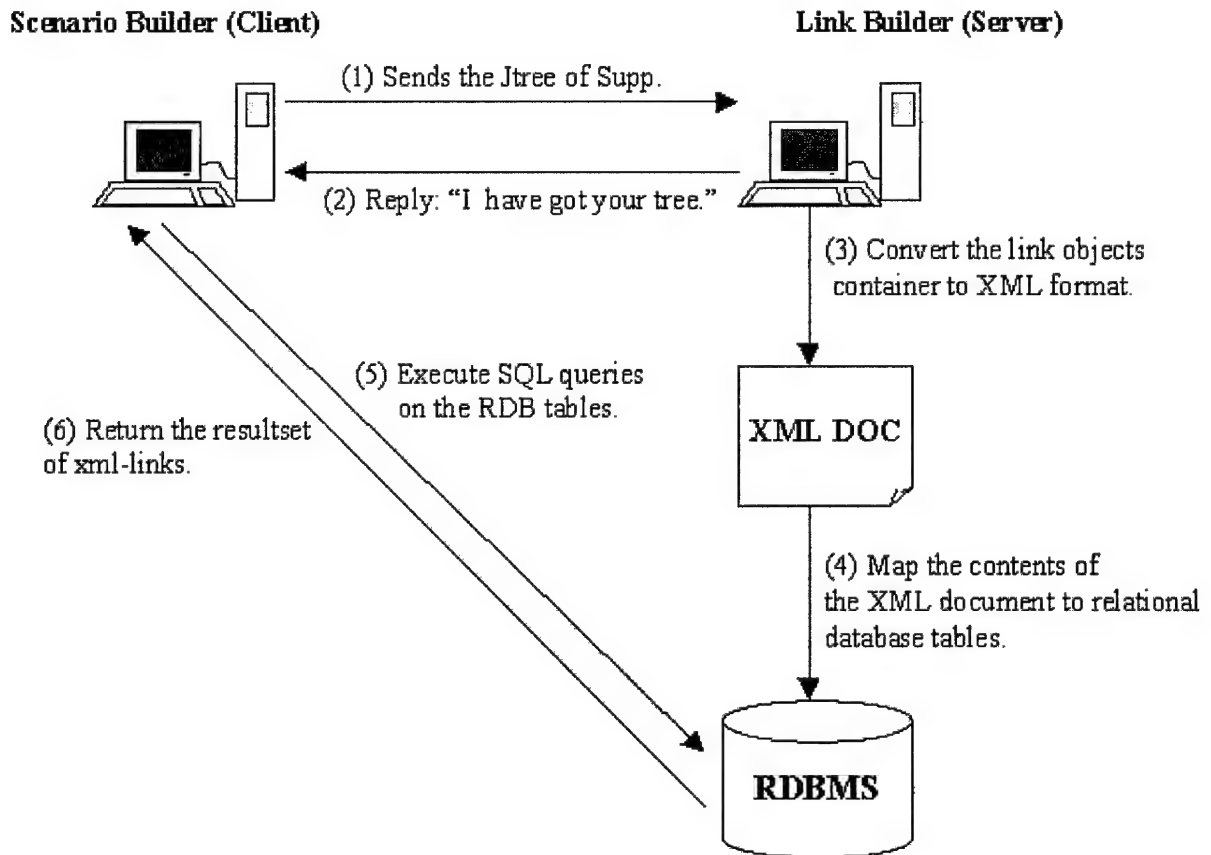

Figure 91- Mapping contents of an XML document to RDB.

When the generically written XML document is read in by the XML parser, the DOM API provides a default tree-like object model. This default object model allows one to extract and modify the data stored in XML documents. If the user decides to use Simple API for XML (SAX) to extract data from XML documents, he has to create his own Java object model. SAX does not provide a default object model. An advantage with SAX could be more flexibility, because the user could create a more efficient object model in accordance with his needs.

Persistence by RDBMS helps us solve both of the problems introduced in section 2.1. It is an efficient way to persist the link objects. The scenario builder does not need to be in "wait state" until the link-objects container is sent over, because the mechanism need not send the container to the scenario builder. What is done is: The XML document which has the link information is mapped to a relational database on the server side. To access the database on the server side, the scenario builder makes a connection (Java Database Connection) with the database and by executing SQL queries on the links table in the database it finds the matching xml-link information row. And this row is returned as the result set of the SQL query to the scenario builder. When the scenario builder selects a node to find the xml-links, the SQL query is prepared generically. It attempts to find the xml-link rows in the links table in the relational database on the server (link builder) side by comparing the path array of the selected node to the values of the "path" column of the "links" table. The matching rows are returned as the result set.

The general view of this mechanism is depicted in Figure 92. In this figure the numbers represent the order of the events. When the scenario builder sends the JTree representation over, the link builder sends a confirmation message to let the scenario builder that it has got the JTree. As the scenario builder has got a reply, it does not have to be in wait state anymore. Then the link builder creates the xml-links and converts them to XML format. After that, the contents of the XML document are mapped to relational database tables. The scenario builder can execute SQL queries on the tables and extract xml-link data from the database without registering with the

130

server. The basic features of RDBMS introduced in section 3.2 allow both the scenario builder

and the link builder make effective use of xml-links.

Scenario Builder (Client)   **Link Builder (Server)**

(1) Sends the Jtree of Supp.

(2) Reply: "I have got your tree."

(3) Convert the link objects
container to XML format.

(5) Execute SQL queries
on the RDB tables.

(6) Return the resultset
of xml-links.

XML DOC

(4) Map the contents of
the XML document to relational
database tables.

RDBMS

Figure 92- The general view of the persistence by RDBMS mechanism.

The persistence mechanism by RDBMS helped us solve the problems specific to the

problems presented in section 2.1. However, it has some drawbacks:

1- There is overhead with mapping object model structure to relational database tables.

2- The specific object model mapped to RDBMS in this study is small and it hides most

of the problems that could easily be encountered in a greater object model. If the

object model were more complex, we could have the problem of **"impedance**

mismatch". That is the difficulty in mapping object graphs to relational databases.

The hierarchical relations between objects are not preserved, when the objects are

131

persisted in a two-dimensional table format. Thus, there is a need to add some more columns to tables in order to preserve the hierarchical relation in the object graph.

3- The conversion from object graph to XML document and then from XML document to relational database tables in Figure 92 is optional. The object model could directly be mapped to RDBMS. It was studied to show the mapping of XML documents to relational database tables. Since the XML document created in this study is not complex, some possible problems do not arise. If the XML document were complex, because of the need to have "foreign keys" to preserve the relations between tables in RDBs, there could be some difficulties when mapping the XML document's content to relational database tables. This difficulty is best expressed in [54]:

"XML is a mismatch with relational databases. You can do tricky joins associating XML type to a database row to make them work, but they are hard to maintain."

The last persistence technology studied in this work is OODBMS. How this technology fits into the problem domain and how it handles all the problems encountered by the two previous approaches are introduced in the next section.

4.3 Persistence by Object Oriented Database Management System (OODBMS)

In this approach the link container object is stored in OODBMS, without any need to create tables as in RDB, the state of the link container object is stored in OODB by "writeObject" methods which are automatically created for each persistence class. It is possible to execute queries on the persistent object graph and extract information as if they were in-memory objects. Figure 93 illustrates this idea.
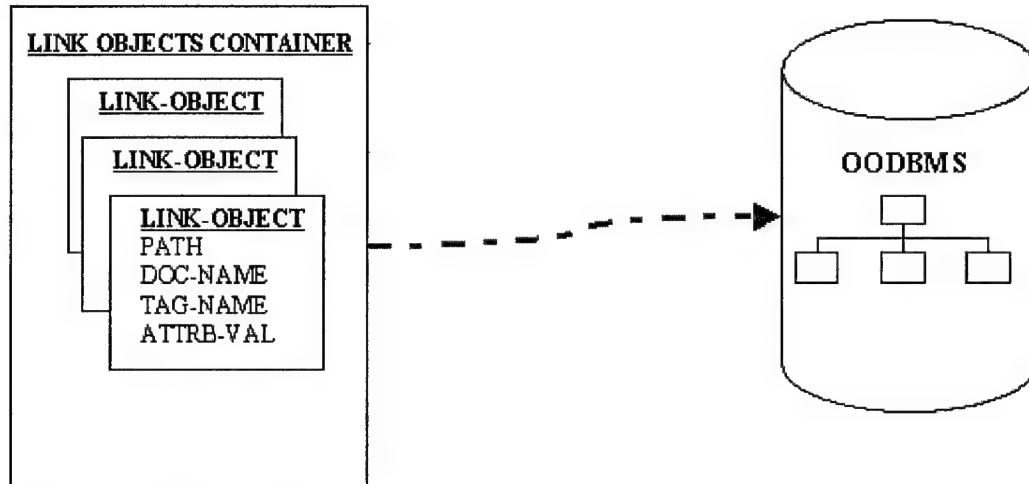
132

Figure 93- Persisting link objects container in OODBMS.

In order to store the link objects container in the OODBMS, there is a need to create a root first. This root object serves as the entry point to the object graph stored in the database. When there is a need to manipulate the database or extract information from the database, by using *getRoot()* the appropriate object model stored in OODBMS is found. After that point, it is possible to traverse in the object graph.

This mechanism solves the problem introduced in section 2.1, and in addition to that it does not have the "impedance mismatch" problem as in the case of RDBMS. The state of the whole object graph is stored in OODBMS provided that they are "persistent capable." The general view of this mechanism is illustrated in Figure 94. The numbers in this figure indicate the order of events.
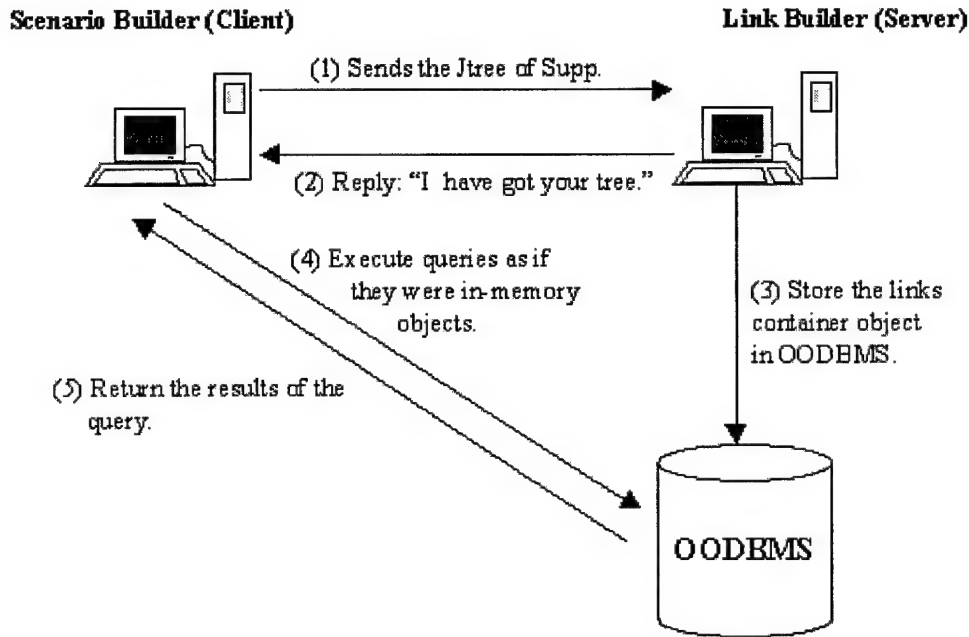
Figure 94- General view of the persistence by OODBMS mechanism.

As shown in Figure 94, after receiving the JTree representation of the Suppressor object from the scenario builder, the link builder sends the "I have got your tree" confirmation reply. Thus the scenario builder does not have to be in "wait state" until all the links are stored in OODBMS. Then the link builder creates the link objects and they are put in the links container object. The links container object is stored in an OODBMS on server side. To access the xml-link information stored in OODBMS, the scenario builder need not register with the server (link builder) anymore. The scenario builder can make a direct connection with the OODBMS, and interact with the links container object graph stored in the OODBMS.

5. Conclusions

When all the three persistence mechanisms were implemented in this study, it was validated that the persistence by OODBMS is the most efficient one. It allows one to store objects in the database, and access and manipulate the objects as if they were in-memory objects. It does not have an overhead like mapping the objects to database tables as in RDBMS. Contrary to serialization mechanism, it allows dynamic queries and dynamic changes on the persisted objects.

In this study, it has also been demonstrated that the data encapsulated in an object model can be written out as an XML document. After that, the contents of the XML document is extracted via DOM API and mapped to relational database tables. In fact, this step was optional and it was presented to introduce the interaction between XML, DOM API and RDBMS.

BIBLIOGRAPHY

1. Simon St. Laurent and Ethan Cerami. <u>Building XML Applications</u>, McGraw Hill Professional Publishing, 1999.

2. <u>Professional Java Server Programming,</u> Wrox Press, December 1999.

3. JP Morgenthal. <u>Enterprise Application Integration with XML and Java</u>, Prentice Hall PTR, 2000.

4. Sean McGrath. <u>XML by Example</u>, Prentice Hall PTR, 1998.

5. David Megginson. <u>Structuring XML Documents</u>, Prentice Hall PTR, 1998.

6. Ian S. Graham and Liam Quin. <u>XML Specification Guide</u>, Wiley Computer Publishing, 1999.

7. Eric Severson. "Approaching Knowledge Management From a Practical Perspective," <u>IBM Global Services EDM Solutions</u>, January 1999.

8. Paolo Ciancarini. "Managing Complex Documents Over the WWW: A Case Study for XML," <u>IEEE Transactions on Knowledge and Data Engineering</u>, Volume 11, No. 4, July-August 1999.

9. Shukri Wakid, John Barkley and Mark Skall. "Object Retrieval and Access Management in Electronic Commerce," <u>IEEE Communications Magazine,</u> Volume 37, No. 9, 1999.

10. Kelli Wiseth. "Introduction to XML," <u>Oracle Magazine</u>, Volume 14, No. 1, January-February 2000.

11. Jeremy Allaire. "Java, XML and Web Syndication," <u>Java Developer's Journal</u>, Volume 4, Issue 8, August 1999.

12. Kelli Wiseth. "XML: Lingua Franca for B2B," <u>Oracle Magazine</u>, Volume13, No. 6, November-December 1999.

13. Harold Treat. "Plugging in to XML," <u>DB2 Magazine</u>, Volume 4, No 4, Winter 1999.

14. Davis Ritter. "The Missing Link for B2B E-Commerce," <u>Intelligent Enterprise</u>, Volume 2, No. 7, May 1999.

15. Ajit Sagar. "Java and XML in the World of E-Commerce," Java Developer's Journal, Volume 4, Issue 6, June 1999.

16. Israel Hilerio. "XML and Java," XML Focus, Volume 4, Issue 9, September 1999.

17. Core Java Volume 1, The Sun Microsystems Press, Java Series, 1999.

18. J. Todd McDonald. "Agent-Based Framework for Collaborative Engineering Model Development," MS Thesis, Air Force Institute of Technology (AU), Wright Patterson AFB, OH, AFIT/GCS/ENG/00M-16.

19. Simon St. Laurent. "DTDs and XML Schemas," www.xml.com/print/1999/12/dtd/index.html

20. "An XML Data-Binding Facility for the Java Platform," www.java.sun.com/xml/docs/bind.pdf

21. JonBosak and Tim Bray. "XML and the Second Generation Web," www.sciam.com/1999/0599issue/0599bosak.html

22. "The Extensible Markup Language (XML)," www.sgml.u-net.com/xml.htm

23. Eric Armstrong. "Code Fast, Run Fast with XML Data Binding," www.java.sun.com/xml/docs/binding/DataBinding.html

24. Todd Freter. "XML: Document and Information Management," www.sun.com/980908/xml

25. Norman Walsh. "A Technical Introduction to XML," www.nwalsh.com/docs/articles/xml/

26. "XML: Enabling Next-Generation Web Applications," Microsoft Corporation, www.msdn.microsoft.com/xml/articles/xmlwp2.asp

27. Bruce Martin. "Lowering the Bar of DOM API," http://www-4.ibm.com/software/developer/library/jguru-dom/index.html

28. "An Introduction to XML," www.personal.u-net.com/~sgml/xmlintro.htm

29. "Extensible Markup Language 1.0," www.w3.org/TR/1998/REC-xml-19980210.html

30. Jonathan Robie. "What is the Document Object Model?" www.w3.org/TR/REC-DOM-Level-1/introduction.html

31. Jon Bosak. "XML, Java, and the Future of the Web," www.metalab.unc.edu/pub/sun-info/standards/xml/why/xmlapps.htm

32. Jon Bosak. "Media-Independent Publishing: Four Myths about XML," www.metalab.unc.edu/pub/sun-info/standards/xml/why/4myths.htm

33. "XML and the DOM," www.java.sun.com/xml/docs/tutorial/dom/index.html

34. Norman Walsh. "Pulling the Pieces Together," www.xml.com/pub/98/10/guide4.html

35. Steven J. DeRose. "XML Linking," www.stg.brown.edu/~sjd/xlinkintro.html

36. "XML Linking Language (XLink)," www.w3.org/TR/xlink

37. JP Morgenthal. "Portable Data / Portable Code: XML and Java Technologies," www.java.sun.com/xml/ncfocus.html

38. Pontus Norman. "A Study of XML," 1999.

39. "XML Path Language," 1999, www.w3.org/TR/xpath

40. "XML Pointer Language, 2," www.w3.org/TR/WD-xptr

41. "XSL Transformations," www.w3.org/TR/xslt.html

42. Michael Talbert and Todd McDonald. "Legacy Scenario Information Reusability for Simulation Interoperability," 2000.

43. eXcelon™ Web Site, www.excelon.corp.com/products/excelon/excelon_toolbox.html

44. Core Java Volume 2, The Sun Microsystems Press, Java Series, 1999.

45. ObjectStore Java Tutorial Release 6.0, Object Design Inc., March 1999.

46. www.faq.oreilllynet.com/XML/TFAQ17.shtm

47. John C. Martin. Introduction to Languages and the Theory of Computation, McGraw-Hill Series in Computer Science, Second Edition, 1997.

48. "XML-QL: A Query Language for XML," www.w3.org/TR/1998/NOTE-xml-ql-19980819/

49. Microstar Software Ltd., www.microstar.com

50. DeLoach, Scott A. "Multiagent Systems Engineering: A Methodology and Language for Designing Agent Systems." Proceedings of a Workshop on Agent-Oriented Information Systems (AOIS'99). 45-57. Seattle, WA. May 1, 1999.

51. DeLoach, Scott A. "Using agentMom." Unpublished document. Air Force Institute of Technology (AU), Wright-Patterson AFB, OH. October 1999.

52. Ajit Sagar. "XML, RDBMS and OODBMS: Peaceful Coexistence?" XML Journal, Volume 1, Issue 2, 2000.

53. Andrew Hunter. "Relational Database Management Systems,"
www.osiris.sunderland.ac.uk/ahu/rds/lec2.html

54. Edmund X. Dejesus. "XML Enters the DBMS Arena,"
www.computerworld.com/cwi/story/0,1199,NAV47_STO53026,00.html, October 2000.

VITA

Teoman Yoruk was born in Ankara, Turkey, in   . He graduated from Ankara Private High School in 1991. His military education started at the Turkish Air Force Academy, Istanbul, in September 1991. He graduated from the Air Force Academy with a B.S. degree in Computer Engineering as a 2nd Lt. in August 1995. Then he was assigned to Air Technical Schools Command Izmir, Turkey. After a training of a year, he was assigned to 4th Main Jet Base, Ankara, Turkey as a logistics officer. In July 1999, he was assigned to Air Force Institute of Technology, Wright-Patterson AFB, Dayton, OH to complete a Master of Science degree in Computer Engineering. He studied Database Systems. His next assignment is at Turkish Air Force Headquarters, Ankara, Turkey.

REPORT DOCUMENTATION PAGE		Form Approved OMB No. 0704-0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing the burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE (DD-MM-YYYY) 20-03-2001	2. REPORT TYPE MASTER'S THESIS	3. DATES COVERED (From – To) JUN 00 - MAR 01

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
MODELS FOR DATA SOURCE TRACING WITH XML	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
TEOMAN YORUK, LT, TUAF	5e. TASK NUMBER
	5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 P. Street, Bldg 640 Wright Patterson AFB, OH 45433-7765	8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GCE/ENG/01M-05

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory, Sensors Directorate Attn: Foster R.M. Bldg 620 S1D34 2241 Avionics Circle Wright Patterson AFB, OH 45433-7303 DSN: 785-2811 x4364	10. SPONSOR/MONITOR'S ACRONYM(S) AFRL/SNZW
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for Public Release, Distribution Unlimited

13. SUPPLEMENTARY NOTES

Advisor: Major Michael L. Talbert, DSN 785-6565 x4280, michael.talbert@afit.edu

14. ABSTRACT

The Air Force Research Laboratory, Sensors Directorate, Electronic Warfare Simulation Branch (AFRL/SNZW) is responsible for developing and maintaining real-world and hypothetical scenarios for an array of threat engagement simulation systems. The general process for scenario creation involves mapping from real-world databases and operations plans to specific fields in the input files which represent the scenario.

As part of the AFRL/SNZW's overall initiative for the development of a Concurrent Engineering Real-Time database CORrelation tool (CERCORT), as important to the scenario files themselves is the capability to trace back to the source of data for the scenario fields. Acquiring this capability consequently results in the reusability of the old scenario components.

In this research, a nascent markup technology, eXtensible Markup Language (XML) and its derivative languages are studied as a basis for representing and capturing the source of data for the fields of an old scenario file and exploiting it for the creation and editing of new scenario files.

15. SUBJECT TERMS

Data Source Traceability, XML, XSLT, DOM, SAX, JTree, Suppressor, TDB, OODBMS, Multi-agent

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON Major Michael L. Talbert
a. REPORT Unclassified	b. ABSTRACT Unclassified	c. THIS PAGE Unclassified	UU	155	19b. TELEPHONE NUMBER (include area code) DSN 785-6565 x4280

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18

141